The Limits of a Decoupled Out-of-Order Superscalar Architecture

Graham P. Jones

Doctor of Philosophy
University of Edinburgh
1999

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Graham P. Jones*)

Abstract

This thesis presents a study into a technique for improving performance in out-of-order superscalar architectures. It identifies three technological trends limiting superscalar performance; they are the increasing cost of a main memory access, control dependencies and the greater hardware complexity of out-of-order execution.

Decoupling is a technique that can provide higher performance through the mechanism of dynamically reordering, asynchronous instruction streams. It offers the capability to improve ILP, through effective latency hiding and dynamic scheduling, and to reduce hardware complexity, through decentralised logic. This thesis evaluates this capability, by investigating the effectiveness of decoupled out-of-order superscalar architectures.

This thesis identifies the degree to which operations can reorder (the degree of reordering) as the critical dimension to an out-of-order superscalar architecture. It investigates the effectiveness of decoupling by focusing on those design issues that determine the degree of reordering, and relaxes all other architectural constraints. This approach allows us to establish the limitations of decoupled out-of-order superscalar architectures.

This thesis shows that a decoupled architecture, through its dynamically reordering instructions windows, provides a possible solution to the problem of latency hiding and issue logic complexity. This thesis demonstrate that for large memory latencies, a decoupled architecture with 2 instruction streams is less sensitive to increases in memory latency than a conventional single stream superscalar architecture. The results also show that for memory latencies greater than 20 cycles, a decoupled architecture can achieve a higher speedup than a conventional superscalar architecture with twice the individual window sizes of a decoupled unit. An explanation for this effect is provided through the concept of the Effective Window Size.

The thesis also investigates a 3-stream decoupled superscalar architecture, that provides dedicated hardware support for resolving control dependencies. The results show that for the partitioning algorithm used in this thesis, the load balancing is poor and the extra hardware resources are under utilised. For this reason the majority of the thesis focuses on a 2 stream decoupled architecture.

Acknowledgements

I dedicate this thesis to my wife Kate, our children Jack, Joseph and Rebecca. I would also like to dedicate it to my mother and father, and to Kate's parents. Unfortunately, this work does not do justice to the time and inconvience it has caused them over the years; in this sense the thesis fails, in that their efforts cannot be read in the lines.

I would especially like to thank my supervisor, Nigel Topham, without whose advice and support these acknowledgements would never have been written.

I would also like thank my fellow office mate and friend, Steve Cussack for helping me stay sane during the long days in 3419. In all that time I can not recall a cross word coming between us.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis presents a study into a technique for improving instruction level parallelism (ILP) and reducing the hardware complexity of out-of-order execution in high performance superscalar architectures. Since the mid-80's we have witnessed a rapid increase in processor performance (on average about 35% per year [45]). To sustain this rate of increase will require innovative techniques capable of detecting greater levels of parallelism and hiding the effect of deleterious technological trends. This thesis explores the application of one such technique, the decoupling paradigm, in superscalar architectures. Decoupling is a technique that provides high ILP and latency tolerance through dynamically reordering, asynchronous instruction streams.

This chapter identifies the technological trends that represent major bottlenecks to superscalar performance. It describes how decoupling could be a potential solution to these problems and outlines the methodology used in the thesis.

The current generation of superscalar architectures, the Alpha 21264 [42],PA-RISC [27],PowerPC 604 [76],SPARC-64 [85],MIPS R10000 [91], all achieve high performance through a hardware scheduling mechanism. This mechanism dynamically resolves data dependencies, issues operations out-of-order and executes multiple operations per cycle. Decoded instructions are placed in a buffer known as the instruction issue window, where they wait until their operands become available. Arbitration logic then selects and issues ready operations to the function units. The maximum number of instructions issued per cycle is known as the issue width; an architecture with an issue width of size n is also referred to as an n-way superscalar machine. To increase ILP in future machines will require larger instruction issue widths and instruction issue windows; this will need greater hardware complexity and increase the length of paths critical to the processor clock speed [64]. To resolve this problem designers are considering the use of decentralised control logic to increase ILP without effecting clock speed [39, 42, 75, 51, 81]. However

this will not solve some of the other bottlenecks to higher ILP.

The exponentially widening gap between memory and processor speeds means that the cost of a main memory access, in terms of missed instruction issue slots, is becoming more expensive. Wulf [89] has referred to the long term consequences of the speed disparity as hitting the memory wall; the point at which making the processor faster produces no performance gain because memory latencies will dominate. Under one scenario Wulf predicts this could occur shortly after the millennium. In addition, the wider issue widths of future superscalar architectures will place greater pressure on the memory system, requiring high sustained bandwidth and low access latency. For these reasons latency hiding techniques such as non-blocking loads [36], stream buffers [50, 77] and data prefetching [18, 16], will remain an important area for architectural research. In the future software and hardware technology will provide greater opportunities for these techniques to optimise memory accesses and hide latencies.

Another bottleneck to detecting parallelism is control dependencies [57]. These dependencies exist between conditional branches and those operations whose execution is dependent on their outcome. Control dependencies limit the amount of reordering that can take place between operations; in [45] the ratio of conditional statements to other operations for the SPEC92 benchmarks is given as 1 in 5 and 1 in 10 for integer and floating point programs, respectively. To resolve this problem designers have used techniques like loop unrolling, branch prediction and speculative execution.

This thesis postulates that the decoupling paradigm could offer a solution to the three fundamental problems:

1. high memory latencies.
2. control dependencies.
3. issue logic complexity.

The decoupling paradigm comprises two types of decoupling, access and control. Access decoupling [37, 72, 80, 90], the most familiar type of decoupling, is a latency hiding technique that partitions - statically or dynamically - a program into two instruction streams to aggressively prefetch data. One stream executed on an address unit (AU) fetches data for the second stream, executed on a data unit (DU). Memory accesses can then be pipelined to tolerate large memory latencies provided the two streams can decouple sufficiently. The decoupling of the streams is also referred to as the slippage [71].

In [9] it was suggested that the decoupling paradigm could be extended to hide the latency of control operations. This technique requires generating an additional stream of operations responsible for resolving control decisions ahead of the computation streams (AU and DU). This stream, loosely coupled from the computation streams, executes on a control unit (CU). This technique is referred to as control decoupling.

This thesis investigates, for the first time, a decoupled architecture in which the processing units are capable of n-way out-of-order superscalar execution. This means operation reordering can occur between (inter) and within (intra) instruction streams. The former is supported by decentralised control logic which allows the loosely coupled streams to slip relative to each other. The latter is supported through each unit having separate instruction windows and issue logic. Previous work on decoupling [81, 7, 37, 72] had only considered the use of single issue, in-order processing units with the compiler being responsible for generating compact schedules for each stream. In 1997, Tyson's thesis [81], suggested that future work should investigate the use of superscalar units in a decoupled machine. The major contribution of this thesis is to provide a detailed analysis of such an architecture.

1.1 Thesis methodology

The most critical dimension to an out-of-order superscalar machine's performance is the degree to which operations can reorder. The degree of reordering is the distance, measured in numbers of instructions, between the oldest and earliest instruction visible to the issuing hardware. A machine with a large degree of reordering will have greater opportunities for detecting the independent operations necessary for parallel execution. The degree of reordering will determine the memory latency tolerance and the ILP for a given machine.

The approach adopted in this thesis focuses the experimental analysis on those factors that determine the degree of reordering, and hence the ILP, in an out-of-order superscalar machine. These factors are referred to as the reordering factors. Ultimately the ILP and performance is bounded by the algorithms used in the program; there is little short of restructuring the code that can be done to improve on this limit. The ability to reorder operations and hence increase ILP in an out-of-order superscalar architecture is primarily determined by:

1. The number of operations in the instruction window that are visible to the scheduling hardware. A large, densely packed window offers greater opportunities for lookahead, reordering and detecting the independent operations

necessary for parallel execution.

2. The data dependencies that exist between operations in the window. Data dependencies impose a partial ordering on the execution of operations. The importance of data dependencies has been well documented in the literature [65, 10]. Flow dependencies exist between operations that produce and consume data values. They represent the minimum ordering of the program that preserves program semantics. However false dependencies may be introduced because of the complexities of memory disambiguation for array variables [69].

An out-of-order decoupled architecture supports mechanisms for inter and intra stream reordering. This extra complexity introduces additional reordering factors that limit the slippage between streams. These factors are

1. The communication traffic and bandwidth between streams. Large levels of communication increase the data dependencies between streams, limiting slippage.

2. The frequency of synchronisation points, also known as loss of decouplings (LOD), between the instruction streams.

Other architectural issues, such as the size of the register file and the available memory bandwidth, are also important, but the reordering factors listed above will determine the maximum amount of ILP. As a limitation study, this thesis only considers those design issues that influence the reordering factors and relaxes all other architectural constraints [1].

The advantage of this approach is that it allows the thesis to focus on the limits of the decoupling paradigm, rather than, as in previous studies [37, 72, 80, 90], a specific implementation of a decoupled architecture. A disadvantage of this approach is that it could be criticised for being too idealistic. However, given the scope of previous work on decoupling [37, 72, 80, 90] and the technological trends described above, the time is right for a quantitative analysis of the limits of decoupled out-of-order superscalar architectures.

1.2 Thesis structure

The narrative of this thesis is presented in the following way. Chapter 2 provides a contextual background to the thesis, reviewing published work on superscalar

[1] However, issue widths were constrained to within projected future values for next generation superscalar architectures.

architecture design and latency hiding technology. It also discusses previous work on decoupling and summarises the original contribution of this thesis.

Chapter 3 describes the three out-of-order superscalar architectures modelled in this thesis. The architectures are characterised by the number of instruction streams executed on the superscalar units. It describes a 3-stream and 2-stream decoupled architecture as well as a typical single stream architecture.

Chapter 4 provides a detailed description of the simulation technique, benchmark programs, software tools and experimental methodology. It explains how correctness of the tools was verified by comparing their output with a simulator used in previous published work. This chapter also describes the three architectures modelled in the thesis.

The remaining chapters present the experimental results and findings. Chapters 5 and 6 provide the groundwork to thesis, developing an understanding of the behaviour and characteristics of the out-of-order decoupled architectures. Chapters 7, 8 and 9 draw on this work to present the key findings of the thesis.

Chapter 5 describes a limitation study into access decoupling under maximum ILP and unlimited resources. The purpose of this study is to ascertain whether decoupling is an effective latency hiding technique for large ILP. This chapter introduces a latency hiding model to enable comparisons between different latency hiding techniques. The experiments show control decoupling to be impractical because of it's under-utilisation of the CU. This chapter discusses how the decoupled architecture and simulation technique were revised in the light of the results from the experiments. **These results were presented at the EuroPar'97 conference [49].**

Chapters 6 extends the work of the previous chapter by considering more realistic architectural constraints. It investigates the effect of superscalar issue width and identifies balanced configurations of AU and DU issue width that maintain maximum throughput.**These results were presented at ParCo97 [48].**

In Chapter 7 a series of simulations are used to investigate the effect of different levels of compiler and hardware complexity in extracting latent parallelism in a decoupled architecture. This chapter demonstrates the importance of architectural support for out-of-order memory accesses and describes how such a mechanism could be implemented in a decoupled architecture without increasing instruction issue logic complexity.

A single instruction stream, out-of-order superscalar architecture could in principle achieve the same level of reordering as a decoupled out-of-order superscalar machine. Chapter 8 provides a qualitative argument for the use of

decoupling through the notion of the effective single window. This chapter substantiates this notion by using an analytical model and simulation studies to compare performance. It shows that for large latencies there is a significant performance gain from decoupling. The single stream machine is shown to require significantly larger instruction issue windows to achieve equivalent performance. **These results were presented at MICRO-30 [47]**

The experiments in Chapter 8 relax those design issues particular to decoupled architectures (see Section 1.1 above) in order not to restrict the slippage between streams. Chapter 9 explores the sensitivity of the previous chapter's findings under more realistic conditions for these design issues. Drawing on the findings from Chapter 7, it also examines the sensitivity of the results to memory ordering.

Chapter 10 summarises the major findings of this thesis, discusses their implications in the context of related research and concludes with a discussion of suggested future work.

Chapter 2

Background

The potential to overlap the execution of independent instructions is known as instruction level parallelism. Multithreaded [31], VLIW [22, 6] and Superscalar [42] architectures all try to improve performance by exploiting latent ILP. Since the early 1990s out-of-order superscalar [1] architectures have come to dominate single processor high performance computing. In 1995, superscalar technology entered the mass production world of personal computing, with the release of the Pentium Pro [8]. Given the historical success and expertise that has been built around superscalar architectures, it is believed that they will continue to be one of the foremost principles of single processor designs. Section 2.1 describes the typical execution of a superscalar machine with reference to current architectures.

Processor performance since the 1980's has grown exponentially at 35% [45] a year. If that rate of increase is to be sustained, superscalar architectures will have to be designed with faster clock speeds and higher ILP. This will require machines with larger issue widths and a larger degree of reordering (see Section 1.1). However, unless some of the complexity for scheduling operations is migrated to the compiler, this extra complexity will introduce delays critical to the processor clock speed. Section 2.4 describes some of the current techniques being considered to reduce the complexity of future out-of-order machines.

Higher performance is also limited by the exponentially widening gap between memory and processor speeds. This gap means that the cost of a main memory access in terms of missed instruction issues slots is becoming more expensive. Wulf [89] has predicted that without technological shift, future processors are in danger of hitting the memory wall. Section 2.2 explains the notion of the memory wall and outlines some of the techniques currently being used to hide memory latency.

[1] This work is only concerned with out-of-order superscalar machines so the prefix out-of-order will not be used in the rest of the thesis

Chapter 1 argued that the degree of reordering was the critical dimension to the performance of a superscalar machine. Section 2.3 presents some of the software and hardware techniques used to increase the degree of reordering.

The author believes that the decoupling paradigm in superscalar architectures could increase the degree of reordering and tolerate large memory latencies. Section 2.5 discusses previous work on decoupling, identifies the benefits of the technique and describes this thesis' original contribution.

2.1 A typical superscalar architecture

Figure 2.1 [2] shows the model of execution for a typical superscalar machine. Instructions are initially fetched from a cache using branch prediction techniques [59] to reduce the effect of control dependencies. In the next stage, instructions are decoded and their operands renamed to eliminate false register dependencies. Register renaming can be implemented using a mapping from physical to logical registers: as used in the Alpha 21264 [42] and the MIPS R10000 [91]. Renaming can also be implemented using the reorder buffer, as in the PowerPC 604 [76]. After renaming, the decode logic dispatches instructions to the issue window, where operations wait for their operands to become available.

Wake-up logic broadcasts the results of completed operations, and once an operations operands becomes available, it is flagged as ready for execution. Arbitration logic is then used to issue ready operations to the execution units. To reduce the impact of control dependencies many superscalar machines such as the Alpha 21264 [91] now implement speculative execution. Speculative execution is a technique that predicts an operation will be executed even though control dependencies are unresolved [84].

While operations may be issued and speculatively executed out-of-order, the machine state must be updated in-order. This process is known as committing the instruction. An instruction is committed, once it is known that it would have have executed if the program had been executed sequentially. The reason why instructions are committed in-order is to ensure a precise state, if an interrupt should occur. This can be implemented using techniques such as checkpointing [85].

[2] This diagram has been taken from Palacharla [77]

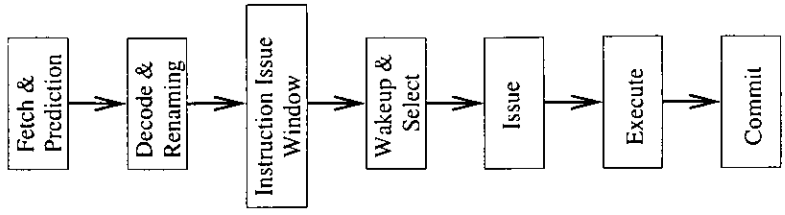


Figure 2.1: Model of execution for a typical superscalar machine

2.2 Latency hiding technology

Wulf [89] developed a simple model of program execution that neatly conveyed the critical importance of memory latency to machine performance. Wulf's model used the equation for the average access time t_{avg} to a data cache

$$t_{avg} = \alpha * t_c + (1 - \alpha) * t_m$$

where α is the hit ratio, and t_c and t_m are the cost of an access to main memory and cache, respectively. Since, typically 1 in 5 operation are memory accesses, when t_{avg} is greater than 5 cycles the memory latency will dominate the critical path; this point Wulf poetically refers to as hitting the memory wall. Wulf predicts from the base year of 1994 that even with a hit ratio of 99.8%, the memory wall would be reached in 10-12 years.

Processor and memory performance are known to be growing at 35% and 7%, respectively [45]. Using Wulf's model Figure 2.2 shows the predicted average access times from the base architecture, the Alpha 21164; the base year is 1995 and the characteristics of the Alpha 21164 [19] are used as an example of a high performance architecture from that year. The 21164 had a clock frequency of 300 MHz and a main memory latency of 253ns [3]. Figure 2.2 also shows that when t_{avg} is 60 cycles [4]; it can be seen even with hit rates of 95%, average access times could reach 60 cycles in the year 2005. Wulf's paper and Figure 2.2 clearly show that latency hiding technology will continue to be a pivotal component of future computer architectures.

There are two types of latency hiding techniques, latency tolerance and latency reduction. Latency tolerance techniques try to overlap access and data operations in order to hide latencies. Latency reduction techniques use memory hierarchies to store frequently used data close to the processing unit.

[3] Interestingly, the latency for a single DRAM component was only 60ns, but the additional overhead of the memory controller, multiplexing addresses to DRAM and the time to drive the DRAM pins increased the latency by a factor of 4

[4] This is the value used in the majority of the simulation studies in this thesis

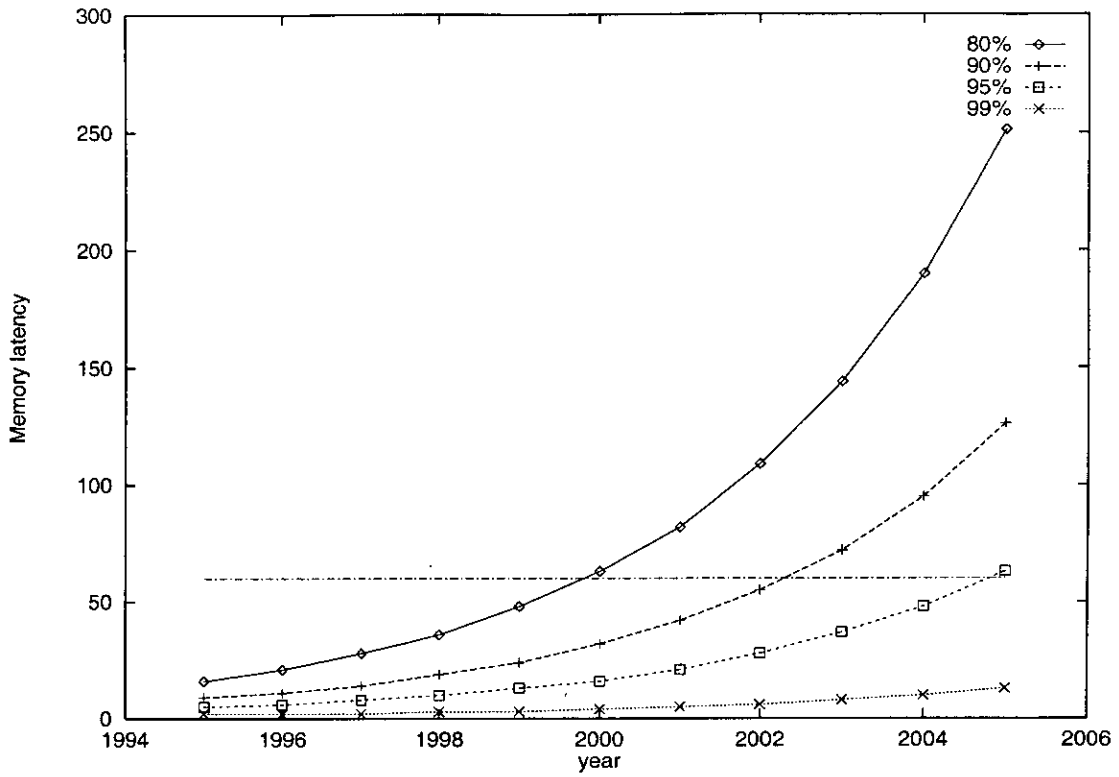


Figure 2.2: Predicted average memory access times for a range of hit ratios (expressed as percentages)

The cache is the most widely used and successful latency reduction technique. It exploits spatial and temporal locality in order to hide the main memory access time. Superscalar architectures with large issue widths and extensive reordering require support for non-blocking loads and multiple accesses per cycle. Non-blocking loads [74, 36, 30] with lock-up free caches [53] reduce miss penalties by servicing accesses while misses remain in-flight.

Cache technology is also being driven by the need to provide high bandwidth multi-ported caches. A multi-ported cache can be implemented by maintaining multiple copies of the data [42], though to ensure coherence writes must be performed in all banks of the cache. The disadvantage of this technique is the large amount of die space required to duplicate the data. Another multi-ported implementation is the interleaved cache used in the HP PA-8000 [27] and MIPS R10000 [91]. Under this scheme accesses to different banks can be serviced simultaneously but conflicts can occur for poor access patterns. However, Rivers *et al.* [68] argued that interleaved banking offered the best cost/performance for a multi-ported cache. The frequency of bank conflicts can be reduced through techniques such as access combining [68] and reordering [81].

A scheme for reordering memory references was also proposed in [81] to reduce the effect of bank conflicts. The Alpha 21264 [42] cache operates at twice the clock speed and therefore logically appears as a dual ported cache. However, with growing working set sizes and increasing ILP, the cost of a miss to first and second cache level has grown. To resolve this problem most high performance architectures include latency tolerance techniques such as data prefetching. These schemes allow data to be prefetched so that cache miss penalties can be tolerated.

Data prefetching techniques tolerate memory latencies by overlapping memory and data operations. It can be implemented in either software [16] or hardware [50]. Software prefetching is a compiler driven technique that places non-blocking loads ahead of the actual data reference. The distance between the non-blocking load and the reference is dependent on static analysis of the loop execution time and memory latency [52]. The disadvantage of software prefetching is the extra overhead of the prefetch instructions [17]. To reduce this effect, heuristics can be used to determine those loads that are likely to result in cache miss [62].

Hardware prefetching [50, 77, 18] has the advantage of dynamically prefetching data according to the run-time conditions of the machine. The simplest form of hardware prefetching is to prefetch the next n blocks after a cache [20]. The disadvantage of the hardware scheme is that it increases memory traffic because prefetched data is less likely to be used [17]. To reduce memory traffic techniques have been developed to detect non-unit strides [41] and to filter out unnecessary prefetching using history tables [77].

Results show that software prefetching is less sensitive to increases in memory latency and introduces less memory traffic [17]. In [56] a comparison of software latency hiding techniques found that when memory latencies are large software prefetching outperforms software pipelining and loop unrolling. At low memory latencies, software pipelining outperforms software prefetching due to the overhead associated with the extra prefetch instructions. Although software prefetching does introduce additional overhead by consuming processor cycles, the relative cost is expected to decrease, as issue widths become larger.

2.3 Increasing the degree of reordering

Chapter 1 identified those factors that determine the degree of reordering, namely data dependencies and the number of operations visible to the scheduling hardware. This sections discusses some of the software and hardware techniques that

influence these factors.

Control dependencies limit the scope of operation reordering to a single basic block. For floating point applications the number of operations in basic blocks is typically in the range 1 to 10. This means that for block scheduling ILP is small. The effect of control dependencies has been reported in [57]. Techniques to remove control dependencies can be implemented in hardware and/or software. Branch prediction and speculative execution [59, 84] are commonly used techniques to ensure the instruction pipeline is fully utilised. Compiler directed if-conversion [2], procedure inlining [43] and loop parallelisation techniques [4, 26] allow more operations to be exposed to the reordering hardware. For example, loop unrolling which replicates the body of a loop and transforms the iteration conditions, has been shown to benefit ILP in superscalar architectures [21].

Unresolved data dependencies between memory references force compilers, in the absence of any supporting hardware, to generate conservative code. While compilers can disambiguate scalar references using variable names and equivalence information, array references are difficult to resolve. A range of static data dependency techniques have been developed [5, 92, 87, 67, 14] to disambiguate array references by examining their subscript expressions. However, it has been shown that in scientific applications, a small number of unresolved data dependencies cause large reductions in ILP [65, 10]. One solution is for the compiler to generate conservative and optimised code, and then select the appropriate code, dependent on run-time checks [21, 15].

Dynamic disambiguation is a technique that removes the constraints imposed by statically unresolved memory accesses. Typically this technique works by memory operations being assigned a sequential number in program order. The semantics of the program are preserved by prohibiting loads to reorder before stores that have the same address and lower sequence number. In [3, 72] all loads referencing memory performed an associative compare with active stores held in a queue. However, the weakness of this scheme is that loads can not be reordered before all preceding stores, and stores can not reorder relative to each other.

The difficulty of dynamic disambiguation in an out-of-order machine is that addresses can be only be compared after they have been computed; this may result in loads waiting for stores when no dependence exists between the reference pair. Speculative disambiguation [40, 61] is a technique that allows references to reorder even though load and store references have not been resolved. Franklin [40] proposed a technique known as the Address Reorder Buffer (ARB) that supports multiple accesses per cycle, using distributed bins. References are mapped to

the bins according to their instruction address, and the bins are used to detect memory conflicts using sequence numbers. The ARB is designed to support speculative memory accesses and out-of-order execution of loads and stores. If a load is incorrectly mispredicted to execute before a store, expensive recovery mechanisms roll back the machine state. To reduce the cost of recovery, Moshovos [61] has proposed the use of prediction techniques to detect references that are likely to result in mis-speculation. Interestingly, this paper shows that the cost of mis-speculation will become significant as instruction window sizes increase.

Data flow dependencies impose a partial ordering on the operations in a program. However, scientific programs are written in imperative languages that introduce false dependencies; these dependencies introduce unnecessary ordering between operations. Register renaming [73] and array privatisation [23] can be used to remove false dependencies. Array privatisation is a technique that eliminates memory dependencies for scalars and arrays. In a study on the PERFECT club suite it was found that the Polaris compiler could produce speedups comparable to manually written code, when array privatisation was included in the optimisation suite [34].

2.4 Reducing reordering logic complexity

Increasing ILP in an out-of-order superscalar architecture requires larger issue widths and instruction windows. However, Palacharla [77] has shown that delays in components critical to the processor clock speeds vary quadratically with issue width and window size. In his paper, Palacharla emphasised the point that it is the product of IPC and clock speed that designers should maximise. Increased logic complexity can improve IPC, but it may also introduce longer delays into paths critical to the processor clock speed. For this reason, Palacharla analysed delays in those functions in the instruction pipeline whose complexity grows with increasing ILP. These functions were renaming, wakeup/select and bypass. Palacharla's key findings are described below.

1. The wakeup and select delay varies quadratically with window size and issue width. This delay was found to dominate at 0.35 μm technology, and larger issue width (8-way issue) and window size (64 instructions).

2. The bypass delay varies quadratically with issue width. For 0.18 μm technology the bypass delay dominates at large issue widths (8-way issue) and window sizes (64 instructions).

Palacharla argues there is a need to consider complexity effective superscalar processors; architecture that can facilitate faster clock speeds without reducing ILP. Palacharla proposed a clustered architecture with centralised decoding and steering logic. Each cluster has a set of FIFO instruction queues that receive operations from the steering logic. The instruction window logic is simplified by only needing to monitor operations at the heads of each queue. The disadvantage of microclusters is the overhead of communicating values between register files and the limited number of operations available for selection. Other work on microclusters [42, 30] has explored the payoffs between simplified hardware and communication hardware.

The Multiscalar [39, 12, 75], MISC [81, 80] and PEW [51] share the common goal of trying to achieve high ILP through decentralised control logic. The MISC and PEW architectures differ from the Multiscalar by using decentralised fetch and decode logic. A detailed description of these architectures is postponed until Chapter 10, where they are discussed in relation to the thesis' findings.

2.5 The decoupling paradigm

Access decoupling [7, 37, 58, 72, 71, 24, 70] is an asynchronous data prefetching technique that tries to hide memory latency by overlapping data and memory operations. Central to all decoupled architectures is an address unit (AU) and a data unit (DU) that communicate, between themselves and memory through the use of queues. Each unit has its own stream of instructions and program counter allowing the AU to run ahead of the DU. The degree to which the AU is ahead of the DU is called the slippage. At certain points in a program's execution, the AU is forced to wait on values from the DU; these points are referred to as a loss of decoupling.

All memory operations are executed by a split load instruction. The address unit computes and sends load and store addresses to memory system via the load and store address queues, respectively. The data from a load is returned from the memory, after some latency, and placed in a data queue where it can be fetched by the data unit. The memory latency can be tolerated, provided the address unit can prefetch data sufficiently far ahead of the data unit. To execute a store operation the address waits in a store address queue until it is matched with a computed value from the data unit, when they are sent to memory.

The early decoupled machines like the ZS-1 [72, 71, 24, 70] and PIPE [37] differed in how they split the instruction stream. The ZS-1 had a single instruction

stream with a splitter whereas PIPE had separate instruction caches for the access and execute unit. The ZS-1, unlike PIPE, also included a data cache. These early studies used synthetic loop kernels to look at the performance of the decoupled architectures for different queue sizes and memory latencies. Recently decoupled machines like the DAE [7], MISC [80, 81] and WM [88] have appeared. The DAE and WM differ from the MISC in their emphasis on providing support for structured data accesses; while the emphasis in the MISC architecture is on the use of decentralised control logic to simplify out-of-order execution.

The DAE [7] includes speciâlised hardware for efficient address generation of structured data. Efficient generation of addresses is shown to increase slippage, reduce DU stall time and increasing cache utilisation. The technique is similar to the access mechanism in vector machines and therefore subject to the same problems of data dependency analysis described in Section 2.3.

The WM [88] architecture consists of 12 asynchronously executing components that communicate by FIFO queues. An integer and floating point unit execute 1 instruction in-order every cycle. Stream control units are used to execute vector like operations for storing and reading data. The results show a factor of 2-9 increase in performance over a scalar architecture

Farrens [38] made a comparison between an in-order 2-way decoupled and 3-way out-of-order superscalar architecture using the Lawrence Livermore benchmark suite. His study showed that decoupled architectures outperformed superscalar architectures. The reasons given for the improvement were the dynamic loop unrolling through unit slippage, the register renaming and the dynamic out-of-order execution between decoupled units. However, this study limited the amount of reordering in the superscalar architecture by making the instruction window only large enough to hold one loop iteration and not allowing it to perform branch prediction. The study also did not consider wider issue widths, real applications and large memory latencies.

In another study by Kurian [55] a comparison was made between a decoupled architecture and a cache based uniprocessor. Decoupled architectures were observed to be more sensitive to memory latency than cache based systems especially with large memory latencies ($\approx$ 15 cycles) when the memory becomes a bottleneck. A decoupled architecture with a cache to capture locality was found to be the best configuration.

Tyson, in [81, 80], discusses the MISC architecture. The MISC machine has four asynchronous units each with their own instruction cache and a common data cache. The motivation behind the MISC architecture was to use decen-

tralised control logic to increase ILP. MISC is a decoupled microprocessor that exploits instruction level parallelism by executing streams in-order, on single issue processing elements. The architecture relies on the compiler to schedule operations and to partition the code. In his thesis Tyson explored the use of different algorithms for producing balanced work loads for the 4 processors. However, he concluded that in order to utilise all the processors, perfect dependency and inter-procedural analysis was required. Tyson also made a comparison between MISC and an in-order Alpha architecture. His results showed that a 4 processor, 1-way in-order issue MISC architecture can achieve higher performance than 4-way in-order issue on an Alpha 21164. However, these results were obtained with a non-blocking cache for the Alpha. This means that the Alpha will stall shortly after a cache miss, but MISC can continue initiating memory requests until the cache queues are saturated.

Topham [79] performed a limitation study of the effectiveness of decoupling for a range of compiler optimisations. The study was based on the static partitioning algorithm in the OCTAVE compiler and an analytical model of decoupled execution. From profiling information, estimates of the number of loss of decouplings in a program were used to calculate upper bounds on decoupling efficiency. The results showed that for many of the PERFECT club suite it was possible to hide large latencies. The partitioning algorithm differed from Tyson's work by trying to reduce the frequency of loss of decouplings rather than ensuring a balanced work load. This limitation study was however based on a static model and used conservative data dependency analysis to estimate the number of loss of decouplings.

Decoupling has gained currency in superscalar architectures like the MIPS R10000 [91] and PA 8000 [27]. The R10000 is able to support a decoupled mode of operation through out-of-order execution and a separate access instruction queue. The HP PA 8000 has two 28 instruction reorder buffers, for data and memory operations, and a large interleaved non-blocking cache. Both the R10000 and PA 8000 can decouple address and execute operations even though there is no architecturally visible AU and DU.

2.6 Original contribution

This thesis is the first known study into the effectiveness of out-of-order execution in decoupled superscalar architectures to improve latency hiding and increase ILP. Exponents of decoupling have typically focused on latency hiding and inter stream

reordering; however, Smith [70] in his early paper argued that an additional benefit of decoupling was the simplification of reordering hardware. This point has been picked up by Tyson [81], but his MISC architecture had multiple streams executing operations in-order. Tyson proposed that future work should consider an out-of-order decoupled architecture. The main contributions of this thesis are:

1. To investigate out-of-order decoupled architectures, identifying those design issues critical to their effectiveness at hiding memory latency and increasing ILP.

2. To compare the effectiveness of an out-of-order decoupled and single-stream superscalar architecture.

3. To investigate how the decoupling paradigm in out-of-order superscalar architectures can reduce issue logic complexity without decreasing performance.

2.7 Summary

This chapter has summarised the strategic problems facing future superscalar designers and the techniques currently being considered to address these issues. It has highlighted the competing tensions between extracting higher ILP and processor complexity, and has emphasised the importance of latency hiding technology. In summary, it is believed that the design goal for future superscalar architectures is: to build complexity sensitive [64], large issue width, latency tolerant machines capable of wider degrees of reordering. The following chapters explore whether this design goal can be met using the decoupling paradigm.

Chapter 3

Three Superscalar Architecture Models

This chapter describes the three superscalar architectures modelled in this thesis. The architectures are characterised by the number of instruction streams they can support. Each instruction stream executes on a single out-of-order superscalar unit, and each unit has a separate instruction issue window, register file and function units. Section 3.1 describes the 3-stream and 2-stream (also referred to as a dual stream) decoupled architectures. The former implements both control and access decoupling, while the latter only implements access decoupling. Section 3.2 describes a 1-stream (single stream architecture) in which no decoupling occurs.

The benefit of the multi-stream architectures, is that reordering can take place within (intra) or between (inter) instruction streams. However, this benefit is bought at the price of extra of hardware and software complexity. Inter-unit communication are implemented by queues as in previous decoupled machines [37, 72, 80, 90]. Separate instruction caches and fetch and decode mechanisms are required to ensure the maximum amount of slippage.

Extra software complexity is also required in the compiler to partition the code [1]. The 3-stream and 2-stream architecture both rely upon the static code partitioning obtained from the experimental OCTAVE compiler [79]. The OCTAVE compiler uses a set of heuristics that minimise communication and LODs between instruction streams. A full discussion of how the compiler was integrated with the simulation technique is described in Chapter 4.

Section 3.4 discusses the design issues that effect the reordering factors (discussed in Section 1.1) for each of the architectures. This section also describes how these issues shaped the experimental methodology used in this thesis.

[1] Partition can also be performed in hardware as in the ZS-1 [72, 71, 24, 70]

3.1 Multi-streamed decoupled architectures

The 3-streamed decoupled superscalar architecture is based on the ACRI machine [9] and is illustrated in Figure 3.1. In this architecture three instruction streams execute on the control unit (CU), address unit (AU) and data unit (DU). Each unit maintains sufficient state to execute independently as an out-of-order superscalar architecture. The bulk of the computational work is performed on the AU/DU pair. The CU is capable of computing conditional branches ahead of the AU/DU pair.

Figure 3.2 shows a 2-stream decoupled architecture similar to the ZS-1 [72] and PIPE [37] machines. This architecture differs from the 3-stream by providing no architectural support for the asynchronous execution of control operations. All operations previously executed on the CU are performed on the AU.

The decoupled memory is capable of sending and receiving data from the AU/DU pair and main memory. The AU and DU can both fetch and write data into the decoupled memory but only the AU can access main memory. This chapter avoids an implementation level description of the decoupled memory, instead concentrating on the semantics necessary to support decoupling [2]

A decoupled load to the DU is performed by way of a split instruction executed on the AU and DU. The AU initiates the load by computing the load address and sending it, via the decoupled memory, to main memory. Once the requested data is returned it waits in the decoupled memory until it is fetched by the DU. The AU is also capable of issuing decoupled self loads in a similar manner.

A decoupled store is performed by the AU sending an address to the decoupled memory, where it waits until the DU computes the data. The decoupled store completes when the address and data are paired and sent to main memory. The CU performs its own accesses via the decoupled memory.

The decoupled memory guarantees that Read After Write (RAW) [3] hazards will be resolved, provided the AU and CU sends loads and stores to the same address in program order. Loads to the same address are, however, allowed to overtake each other. The decoupled memory detects RAW hazards by matching load and store addresses. When a hazard is detected the load is suspended until the store completes. This thesis investigates two different decoupled memory models for handling RAW conflicts:

1. In the first memory model each load waits for completion of any outstanding

[2] A full description of the decoupled memory is provided in Section 7.4

[3] The simulation technique (see Chapter 4) can remove false dependencies so that Write After Write (WAW) and Write After Read (WAR) data hazards do not need to be considered.

writes to the same address before going to main memory. This is referred to as the basic decoupled memory since it relies only on the slippage between the AU and DU to hide the main memory latency.

2. In the second memory model, information about future access patterns is used to cache data close to the DU. The load is combined with the matching store so that when a store datum is received, the load can be satisfied without accessing main memory. This is referred to as the optimised decoupled memory.

Figures 3.3 and 3.4 illustrate the effects of using the basic and optimised decoupled memory, respectively. These diagrams use the following notation to describe the times at which operations occur[4];

- s_i^{AU}, s_i^{DU} denote the times at which the AU and DU perform their respective operations in a decoupled split store.
- l_i^{AU}, l_i^{DU} denote the times at which the AU and DU perform their respective operations in a decoupled split load.
- o_i^{DU} denotes the time at which an operation on the DU actually uses the loaded value.
- c_M denotes the execution cost of a main memory access.

In Figure 3.3 the load address is sent to main memory and as a consequence the DU fetch operation has to wait for the value to be returned by the memory system. In Figure 3.4 the DU does not incur any delay because the copy is resident in the decoupled memory. The optimised decoupled memory model also acts as a bandwidth filter, reducing memory bandwidth requirements.

3.2 A single stream superscalar architecture

The single stream superscalar machine is shown in Figure 3.5. The architecture is an out-of-order machine with a single instruction window for reordering operations. In each cycle, independent operations that are ready to execute are issued to the function units. Unlike the decoupled machine the full issue width is available for issuing instructions every cycle.

[4]The subscript denotes the ith decoupled load during the execution, where $1 \leq i \leq N$, when N is the total number of executed decoupled loads.

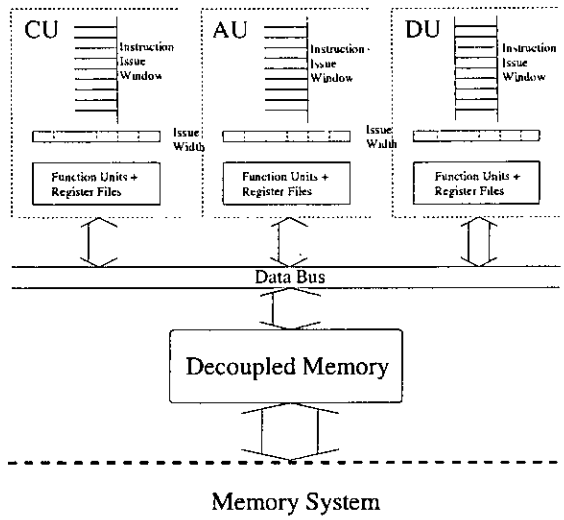


Figure 3.1: A 3-stream decoupled architecture

There are different types of hardware, software and hybrid schemes for data prefetching. The single stream architecture uses a hybrid scheme. Every load operation comprises two instructions, a prefetch and an access operation. The prefetch instruction pre-loads data into the prefetch buffer ahead of the access operation. Prefetch operations, unlike software schemes, are allowed to begin execution as soon as runtime resources allow. This scheme combines the benefits of exact address computation with dynamic execution. The prefetch buffer is a fully associative buffer responsible for storing prefetched data. Requests from the prefetch buffer take 1 cycle.

3.3 The memory system

The memory system for all three architectures consists of the main memory but may also be composed of multi-level caches. A detailed simulation of the memory system is beyond the scope of this thesis; instead its execution is modelled by considering every access to have a fixed cost.

It is recognised that this model is a simplification of the complex memory systems in high performance architectures. However, it is believed that this approach is justified because most memory systems are composed of memory hierarchies and each level in the hierarchy supplies a quality of service to higher levels. The quality of service can be quantified as a constant term, the average access time to that level. Using this simplification has the benefit that it allows the memory system to be modelled with a high or low quality of service. This equates to the processor interfacing with a complex (e.g. having multilevel caches) or basic (e.g.

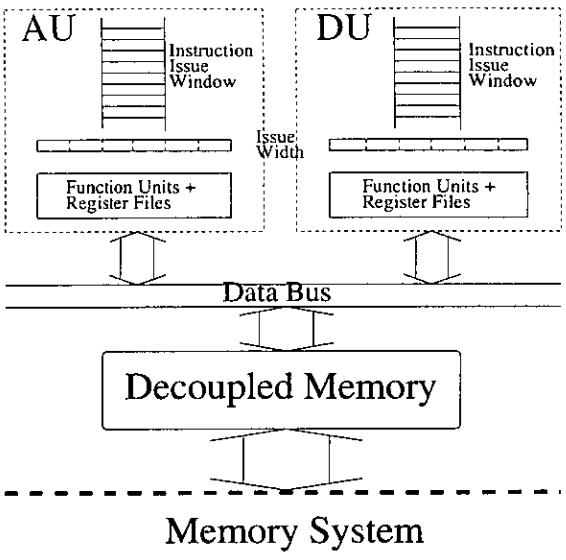


Figure 3.2: A 2-stream decoupled architecture

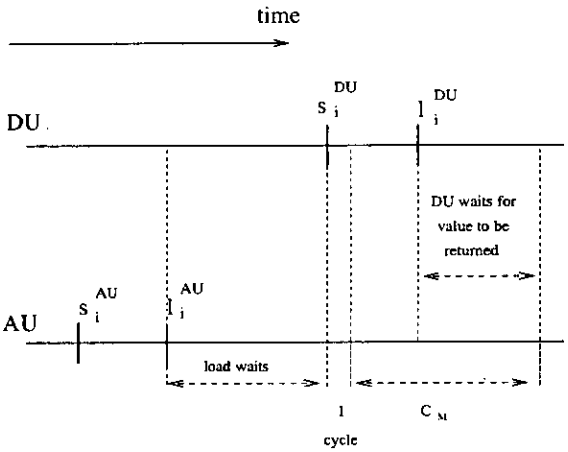


Figure 3.3: Model of the basic decoupled memory

just the main memory) memory systems. It also allows the thesis to concentrate on areas of the design space that have not previously been studied. Much of the previous work on decoupling has concentrated on the size of queues [71] and the use of data caches [55]

3.4 The design issues for the reordering factors

In Section 1.1 it was argued that the reordering factors were the critical determinants of the maximum ILP in superscalar architectures. This section identifies those design issues that influence the reordering factors; these issues are also referred to as the degrees of freedom for reordering.

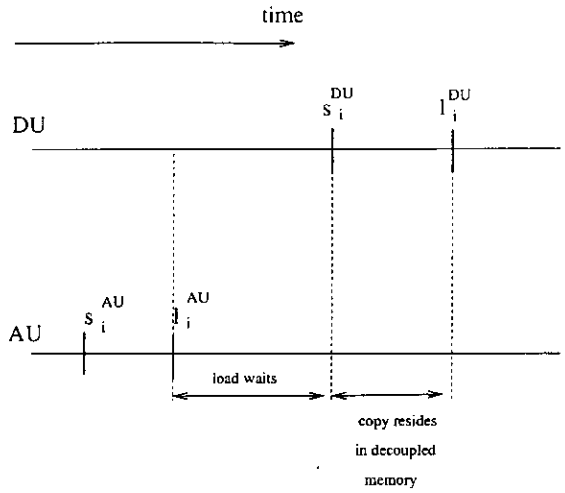


Figure 3.4: Model of the optimised decoupled memory

The degrees of freedom are shown in Table 3.1. The third column shows where they apply to a single stream architecture (SSA) or decoupled architecture (DA). The final column shows the chapter in the thesis where these design issues are considered.

The degrees of freedom are effected by hardware and software. Data dependency analysis can be static [67], dynamic [40] or a hybrid of both [21]. The number of operations in the instruction window is determined by hardware factors such as its size, the use of speculative execution [84] and branch prediction [59]. Software optimisations [63, 4, 26] can also expose more parallelism to the hardware through loop unrolling, inlining and loop transformations.

1.	parallelising optimisations	SSA, DA	ch. 7
2.	memory ordering schemes.	SSA, DA	ch. 7& ch.9
3.	data dependency analysis.	SSA, DA	ch. 7
4.	register and memory renaming.	SSA, DA	ch. 7
5.	synchronisation points	DA	ch. 7
6.	instruction window size.	SSA, DA	ch. 8 & ch. 9
7.	inter-unit bandwidth	DA	ch. 9
8.	partitioning and code expansion	DA	ch. 9
9.	decoupled memory/prefetch buffer capacity	SSA, DA	ch. 9

Table 3.1: Design issues for out-of-order machines

In this thesis the focus of the experiments is on the effect of the degrees of freedom on the superscalar architectures (other architectural constraints were relaxed). The advantage of this approach is that it isolates decoupling from other factors and allows the limits of the technique to be explored. The disadvantage

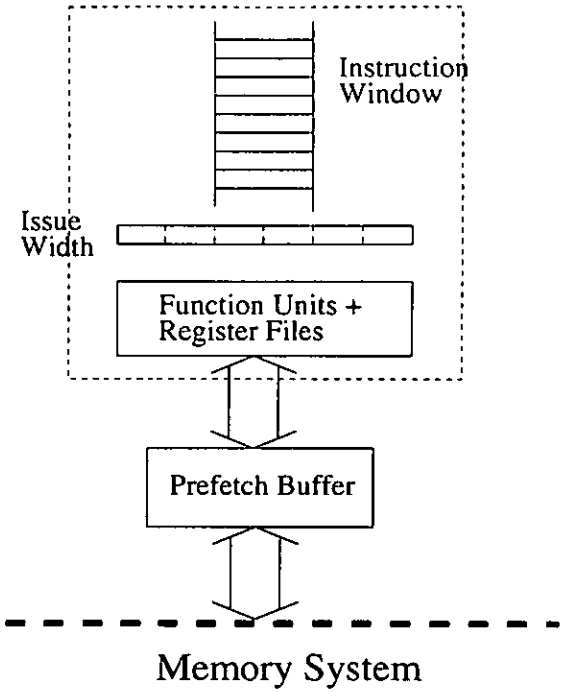


Figure 3.5: Single window superscalar machine

is that it is open to the criticism of being idealistic. Given the scope of previous studies into decoupling (see Section 2.5) and the technological developments discussed in this chapter it is believed the time is ripe for a quantitative analysis of the limits of decoupling.

The approach adopted in the experiments was to initially assume an ideal decoupled architecture where the only constraints were true dependencies, and then to incrementally introduce each of the degrees of freedom from Table 3.1 into the experimental domain.

Chapter 4

Description of the Simulation technique

The aim of the simulation methodology applied in this thesis was to identify performance trends and design issues rather than provide a detailed exposition of a particular system design. This approach was used because previous work [7, 37, 58, 72, 81] had concentrated on the specific implementation details of access decoupling rather than the more generic paradigm of dynamically executing, asynchronous instruction streams. To provide a new perspective into decoupling, it was decided to adopt a methodology that relaxed some of the technological constraints of earlier work.

Modern high performance architectures comprise both sophisticated software and hardware technology; therefore an experiment environment to evaluate a novel architecture should model the design space for both aspects of technology. The simulation tools developed for this thesis were designed to provide an experimental framework capable of modeling a wide range of compiler and hardware configurations. In keeping with the simulation methodology, the framework was also designed to:

1. be independent of any native compiler.
2. model potential future improvements in compiler and hardware technology.
3. to relax technology bottlenecks important for latency tolerance technology. For example, data prefetching depends on finding instruction level parallelism in order to overlap operations and hide memory latency. Yet, extracting instruction level parallelism is known to require sophisticated restructuring and data dependency algorithms [11, 10, 23, 69].

This chapter explains the implementation and operational details of the framework and is developed in the following way. Section 4.1 provides a detail descrip-

tion of the experiment framework. It explains how scientific applications were *annotated* so that execution of the compiled program would generate calls to a *simulator*. The simulator was parameterized to model the execution of the 3 types of superscalar architectures, described in Chapter 3, under a wide range of architectural configurations. Sections 4.2 and 4.3 discusses the advantages and disadvantages of using this approach. Finally, Section 4.4 describes the benchmark programs used in the experiments.

4.1 Experimental framework

The experimental framework uses a technique adapted from work by Petersen [66]. The strength of Petersen's technique was that it provided an environment in which to explore many different compiler issues. This method works by annotating the source level code with calls to routines within an architecture simulator. Shadow variables are inserted into the program to track the earliest time that program values become available. Shadow variables are passed as arguments to the simulator to enable operation start times to be computed. Simulation of the system can then be performed by executing the annotated program.

Although, there were many advantages to using Petersen's technique in the framework, his modelling tools could not be adapted to model out-of-order superscalar execution and therefore were not sufficient for the thesis requirements. For this reason, the entire experimental framework was developed during the course of the thesis. The techniques used to extend Petersen's method are original work.

The two components to the framework, the FOrtran Code Annotation Tool (FOCAT) and Modelling EnviroNment for Decoupled Out-of-order Superscalar Architectures (MENDOSA) are described in Section 4.1.1 and Section 4.1.2.

4.1.1 FOrtran Code Annotation Tool (FOCAT)

This section describes the basic concepts of Petersen's technique and explains how it was extended and developed into a generic annotation tool. Extending the technique involved identifying general annotation rules that allowed FOCAT to be independent of the routines being inserted into the code. This extension facilitated integration with the simulator and enabled FOCAT to be validated by comparison with results from Petersen's annotator.

Basic Concepts

A program can be represented as a data flow graph in which the nodes and arcs symbolize operations and the flow of data, respectively. Running a program can be viewed as the dynamic execution of that data flow graph where the longest path through the graph determines the minimum execution time of the program; this is also known as the critical path length.

Petersen [65] developed a technique that annotated the source code so that running the program modeled the dynamic execution of its data flow graph. Using this approach Petersen was able to evaluate performance upper bounds for different data dependency tests independent of any architectural model. He was also able to use the technique to investigate the speedup from operation, statement and loop level parallelism.

In Petersen's technique the execution of a data flow graph is modeled by using shadow variables to enforce the data dependencies between operations. Shadow variables are time stamps inserted into the program to track the earliest time that program values become available. Each program variable has an associated shadow variable to track it throughout the execution of the program.

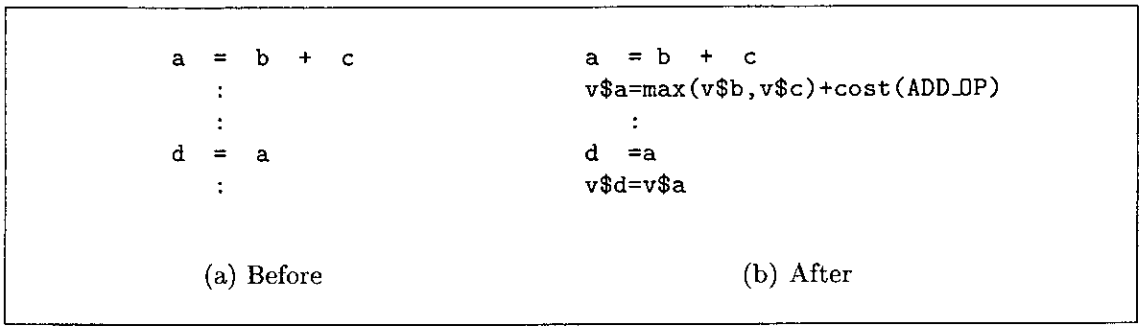
Figures 4.1(a) and 4.1(b) show a small section of Fortran code respectively before and after annotation. Figure 4.1(b) shows the shadow variables denoted with the prefix `v$`, the generic `max` function and the `cost` function. The `cost` function returns the time in cycles to compute the addition of the two operands. When this section of code is executed `v$d` will hold the earliest time that the value in `d` was made available.

The annotation process can be extended to different program constructs such as if statements, procedure calls and do loops. Figure 4.2(a) shows how conditional statements can be annotated to enforce control dependencies between operations. An additional shadow variable is generated `v$if` to hold the earliest time the branch was resolved. The control dependency is enforced by including the shadow variable in the `max` function for all statements that are control dependent on the branch. The `do loop` construct can be annotated in a similar way.

To annotate a subroutine call the parameters' shadow variables are appended to the procedure's argument list. A new shadow variable is also generated and passed as an extra argument to the routine. The new variable enforces dependencies between statements in the callee and called routines. Figure 4.2(b) illustrates how a procedure call would be annotated.

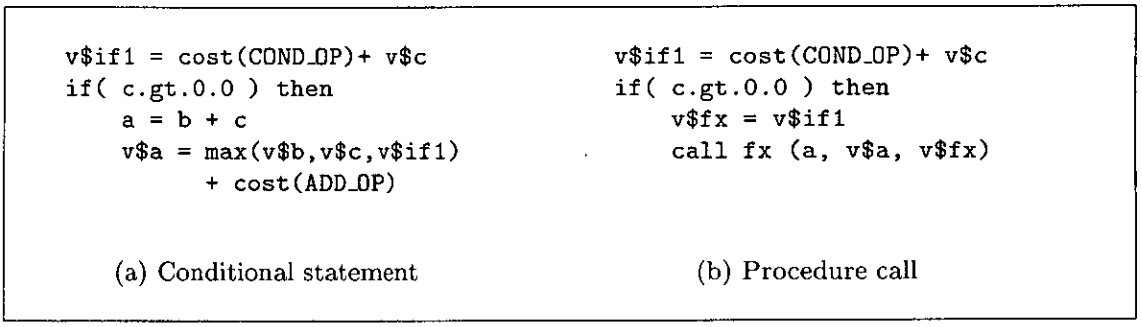
For a full discussion of the technique the reader is referred to Petersen's the-

sis [65]. In order to develop the software tools necessary for the thesis, the technique was extended and adapted into a generic annotator. The process of designing and building FOCAT is described below.



(a) Before (b) After

Figure 4.1: Basic statement



(a) Conditional statement (b) Procedure call

Figure 4.2: After annotation

Designing and implementing FOCAT

FOCAT was written in C++ using an object oriented methodology to promote a readily extensible design capable of meeting later experimental requirements. The tool parsed Fortran code and annotated it with a set of user defined routines. This generic functionality was provided by recognising the similarities between Petersen's annotation technique and the general data flow analysis framework described in [1]. This framework is often used as part of a compiler's analysis suite, and it's most common application is for code optimizations such as copy propagation and detecting loop invariant computations. The framework works by iteratively traversing all the arcs of data flow graph and gathering information about the way data flows through the program. The framework requires:

1. a set of values to be propagated along the arcs of the graph.

2. a confluence function that combines the sets of values from all the arcs incident to a node.

3. for each node operator, a transfer function that maps a set of input values to a set of output values.

4. an update function that updates a set of values.

Petersen's technique is similar to this framework in that execution of the annotated program can be seen as an information gathering process. In his case the information being propagated, via the shadow variables, is the time at which program values become available. The confluence and transfer functions are the `max` and `cost` routines, respectively. The update function was the assignment operator. The framework differs by being a static technique which must follow all paths in the data flow graph while Petersen's technique is dynamic and only follows one path through the graph.

By recognising the parallels between the two techniques it was possible to design the tool based on a set of general rules for annotating Fortran code. Like the data flow analysis framework, the user of the framework only needs to supply a set of confluence and transfer routines to FOCAT. This design had the benefit of hiding complexity and making it extensible to different types of annotation.

For the annotator to interface with the architecture simulator the program had to be annotated with the following routines;

- `idop, idload, idstore`. These routines were responsible for simulating a data operation, a load and a store address computation, respectively. These routines provide the transfer functions.

- `iddef`. This routine was responsible for simulating store operations and for updating shadow variables.

Each of the above routines accepted shadow variables as part of their argument list. The `max` routine (the confluence operator) did not explicitly appear in the annotated code, instead it was implicit in each one of the above routines. For the decoupled architecture, shadow variables had to track the time at which values became available on each of the units. One way to implement this would have been to have three shadow variables for each program variable, but this would have reduced the readability of the code. Instead the program was still annotated with one shadow variable but this was a reference to a structure that contained three timestamps, one for each unit.

Simulating the decoupled architecture required splitting the code between the different units. This was performed statically by the partitioning algorithm in the OCTAVE compiler [79]. The compiler assigned each node in the data flow graph to one or more units. During the annotation process this information was inserted into the argument list for the simulator routines. At runtime this argument was used to execute operations on the designated unit.

The OCTAVE compiler was also used to identify where load operations occurred in a program. Part of the optimization suite for the compiler was to detect redundant loads in the program. This information was used by the annotator to reduce the number of `idload` routines in the code. An example of the code annotation for the superscalar architectures can be seen in Figure 4.3. The code placement information was ignored by the single stream architecture.

```
dy(iy)=(dy(iy)+(da*dx(ix)))
call iddef(v$dy(iy) , DU ,
          idop(DU , ASSIGN_OP, FLOAT ,
              idop(DU , ADD_OP, FLOAT ,
                  idload(DU , ARRAY_LD, FLOAT , v$dy(iy) , v$iy) ,
                  idop(DU , MULT_OP, FLOAT ,v$da,
                      idload(DU , ARRAY_LD, FLOAT , v$dx(ix) , v$ix)
                      )
                  )
              ),
          idstore(DU , ARRAY_ST, FLOAT , v$iy)
          )
```

Figure 4.3: Code annotation for superscalar architectures

Verification of FOCAT

Correctness of FOCAT was verified by replicating Petersen's experiments of critical path times. Both tools were used to annotate the programs with `max` and `cost` functions. The annotated programs were run and the simulated critical path times compared.

Table 4.1 shows the measured critical path times for the two annotators. The results show that for 4 of the programs, DYFESM, FLO52Q, TRFD and MDG the difference between the annotators is marginal. However for four of the programs the differences are large and need explanation. These differences were caused by variations in the implementation of the two annotators. These were :

- an optimization, used in Petersen's tool, to reduce experiment run-times

resulted in longer critical path measurements. The effect of the optimization can be seen in a large compound statement. Consider a line of code

```
x = a + b + c + d
```

If we assume that all operations take 1 cycle Petersen's annotator would annotate the program with

```
v$x = max (v$a, v$b, v$c, v$d) + 4
```

whilst FOCAT would correctly generate

```
v$x = max ( max (v$a, v$b) + 1, max (v$c, v$d) + 1) + 2
```

The difference is caused by Petersen's annotator trying to reduce the number of function calls to `max`. The result is that the computed available time of x is 1 cycle earlier in the code generated by FOCAT. The effect of this is most apparent in QCD2, where the measured critical path time from the code generated by FOCAT is less than that for Petersen.

- Petersen's annotator did not enforce control dependencies in `while` loops. The `while` loop construct is implemented in Fortran by a conditional branch out of a `DO` loop. This construct means a cyclic control dependency exists from one iteration to the next. If this dependency is removed, as in Petersen's annotator, there is more parallelism and hence a lower execution time for the loop. In FOCAT the cyclic dependency is enforced. The net effect of the difference in implementation is for the overall execution time to be greater for FOCAT.

  Table 4.1 shows that programs with `while` loops have longer execution times. The most significant difference is in the program TRACK. This is due to the routine `extend`, which executes for the largest percentage (24%) of the total execution time, having 8 `while` loops in it's body.

4.1.2 Modelling EnviroNment for Decoupled Out-of-order Superscalar Architectures (MENDOSA)

Software and hardware techniques must translate the data flow graph to a target architecture so that all dependencies are preserved. Due to hardware constraints and limitations in compiler technology the translation process introduces additional dependency arcs that cause the critical path time to increase.

In MENDOSA, resource conflicts and compiler restrictions are modeled as additional dependency arcs; in this thesis these are referred to as architectural

Program Name	Petersen's annotator	Annotation Tool	Difference (%)	`while` loops
ADM	0.369	0.410	+11	8
ARC2D	0.485	0.509	+5	2
DYFESM	17.314	17.348	+0.2	2
FLO52Q	1.315	1.312	+0.2	1
TRFD	0.676	0.676	0.0	0
MDG	1.165	1.165	0.0	0
QCD2	17.555	16.334	-7	0
TRACK	0.676	28.767	+6716.8	14

Table 4.1: Critical path times for different annotation tools

dependencies. Like the false dependencies in data flow graphs these dependencies can be removed with sufficient resources. In MENDOSA, operations can only begin execution when all data flow and architectural dependencies are preserved.

The arguments to the interface routines are shadow variables and source level information about operation type and code placement. At runtime this information is used to translate source level operations into single instructions that can be executed in the simulator. The OCTAVE code placement is used by the decoupled architecture to execute instructions on the correct unit.

4.2 Advantages of the simulation technique

This section describes the advantages of the simulation technique implemented in MENDOSA and FOCAT. For all the superscalar architectures discussed in Chapter 3, good dependency analysis is essential for producing high levels of ILP, but for the decoupled architecture it is also important for reducing the number of loss of decouplings between the instruction streams (Chapter 2 explains the phenomenon of a loss of decoupling).

One of the powerful aspects to this simulation technique is that it can model the execution of a program in which the data dependency analysis is perfect; this means that only true dependencies will exist between operations. Perfect dependency analysis can be modeled because of the one-to-one mapping between program and shadow variables, and the `max` function only taking operand shadow variables as arguments.

The technique can also model data privatisation and renaming through the use of shadow variables. This is illustrated in Figure 4.4 where the potential WAR hazard from the read and write of `a[i]` is removed. The hazard is caused by a

name conflict and can be removed by renaming; the annotation technique allows this section of code to be modeled as if renaming had been implemented. If `a[i]` were included in the arguments to `max` this section of code could be modeled as if the false dependency had not been resolved (see Figure 4.5).

The technique can also model loop transformations (i.e. loop unrolling and loop merge) and inter-procedural analysis (i.e. inlining) by introducing additional shadow dependencies. These shadow variables are inserted to carry the times at which loops and procedures finish execution. They can then be used as arguments to the annotated routines to enforce dependencies between high level constructs such as loops and procedure calls. In this way operation level parallelism can be modeled at different loop and procedural levels. Since a wider scope for reordering operations represents increasing compiler sophistication this technique provides a way of determining the accrued benefits of greater compiler development effort.

In summary the benefits of the simulation technique are:

- that it provides a mechanism for simulating a range of compiler optimizations known to be necessary for high instruction level parallelism and latency hiding. These optimizations are
  - data dependency analysis.
  - inter-procedural analysis.
  - loop transformations.
  - data privatisation and renaming.
- the program is simulated at the source level allowing the experimenter to concentrate on the high level semantics of access decoupling without bringing in issues of assembly code generation.
- it remains independent of any particular native compiler.
- correctness of the annotation was verified with reference to previously published work.

4.3 Disadvantages of the simulation technique

The difficulties and criticisms of using this simulation technique are:

- Code expansion due to the annotation routines and shadow variables inserted in the code. For example, FLO52Q code size increased by approximately a factor of 8.

```
x   = y  *  a[i]
v$x = max(v$y,v$a[i])
      + cost(MULT_OP)
    :
a[i] = b[i] - c[i]
v$a[i] = max(v$b[i], v$c[i])
         + cost(SUBT_OP)
```

Figure 4.4: Code without WAR dependency

```
x   = y  *  a[i]
v$x = max(v$y,v$a[i])
      +  cost(MULT_OP)
    :
a[i] = b[i] - c[i]
v$a[i] = max(..., v$a[i])
         + cost(SUBT_OP)
```

Figure 4.5: Code with WAR dependency

- The increase in memory required to run the applications. This is due the whole memory space of the original program being duplicated because of the one-to-one mapping from program to shadow variables. When simulating the decoupled architecture the memory requirements are higher because each unit has it's own timestamp.

- The execution time of the programs. The large number of procedure calls results in much longer execution times for the program. In order to resolve this problem a technique was developed that simulated a representative sample of a program's behaviour. This technique is described in Appendix C.

- It could be argued that a source level simulation of the program removes too much of the complexity inherent in assembly code. However, using assembly code would have tied the experimental findings to a native compiler and would have removed the benefits of modeling a wide range of different compiler optimizations.

4.4 Benchmark programs

A selection of 8 scientific Fortran programs from the PERFECT club suite [25] were chosen as the benchmark applications. These were chosen because they represented a set of real applications composed of a wide range of scientific algo-

rithms. These benchmarks were selected to represent varying degrees of vectorization and known degrees of decoupling. Table 4.2 shows the benchmark programs with their reported proportion of vectorised operations (VO) obtained from [82] and the decoupling efficiency (DE) obtained from [79]. The table also shows the main algorithms used in each program.

In retrospect, multimedia applications would have been equally appropriate as a benchmark suite. However, at the start of this thesis the majority of interest was in the use of high performance architectures for scientific applications. It is probable that future high performance architectures will be significantly influenced by the requirements of multimedia applications. These applications will involve data compression algorithms, 3D graphics, encryption and image processing; all of which comprise some type of vector manipulation or use algorithms such as FFT. There will therefore be strong similarities between the behaviour of multimedia applications and the benchmarks shown in Table 4.2.

Program Name	VO (%)	DE (%)	Algorithms used								
			1	2	3	4	5	6	7	8	9
ADM	43	69	.	.	x	.	.	.	.	.	.
ARC2D	91	99	x	.	.	x	.	.	.	.	.
DYFESM	69	77	x	x	.	.	.	x	.	.	.
FLO52Q	92	82	.	.	.	.	x	x	.	.	.
MDG	88	92	.	.	.	.	.	x	.	.	.
QCD2	4	19	.	.	.	.	.	.	x	.	.
TRACK	14	14	.	.	.	.	.	.	.	.	x
TRFD	70	99	.	.	.	.	.	.	.	x	.

Algorithms :

1.Sparse linear systems solvers 2.Nonlinear algebraic system solvers
3.Fast Fourier Transforms 4.Rapid elliptic problem solvers
5.Multi-grid schemes 6.Ordinary differential equation solvers
7.Monte Carlo schemes 8.Integral transforms
9.Convolution

Table 4.2: Benchmark programs from PERFECT club suite

Chapter 5

The Limits of Decoupling

Studies into access decoupled architectures have shown it can successfully hide memory latency when memory latencies are small, ILP is low and data dependency analysis is conservative [55, 71, 79]. Yet future high performance architectures will be characterised by relatively slower memory speeds and increased ILP. Increased ILP will place greater pressure on memory systems and require higher sustained bandwidth. One can also predict that improvements in compiler technology and data dependency analysis may offer greater opportunities for latency tolerance. Advances in technology will therefore create a tension for latency hiding techniques; on the one hand increasing the pressure on the memory system but also allowing greater operation overlap and latency tolerance. To evaluate the effectiveness of access decoupling in future superscalar architectures one would ideally assess it in relation to future compiler and hardware technology. Such an assessment is clearly impractical. An alternative approach is to consider:

> The effectiveness of access decoupling at tolerating large memory latencies when all the software and hardware constraints to ILP are removed.

This chapter describes a study into the theoretical limits of latency hiding through decoupled execution under maximum ILP. To focus on the limits of access and control decoupling, the 3-stream decoupled architecture has unlimited computational resources and ideal out-of-order execution.

This chapter is organised in the following way. Section 5.1 describes the latency hiding model and introduces β, an architecture independent measure of the efficiency of a latency hiding technique. Section 5.2 discusses the simulation technique and Section 5.3 presents the experiment results. As a consequence of the findings in this chapter, Section 5.4 revises the architecture models used in the experiments. It was observed that some programs had irregular behaviour and

therefore it was decided to simulate them in full. However, because this took an excessive amount of time, only a subset of the benchmark programs were selected from Table 4.2. Appendix C discusses a technique, known as sampled simulation, that reduces execution time and yet captures a program's behaviour accurately. Section 5.5 concludes with a summary of this chapters findings.

5.1 A model for latency hiding

The use of memory hierarchies in high performance architectures is a consequence of the need to balance the cost, capacity and performance benefits of different memory technologies. Latency hiding techniques try to hide the latency of the slowest level of the memory hierarchy so that all accesses are perceived by the CPU to occur at the speed of the fastest element. The difference between the speeds of the fastest and slowest memory is referred to as the memory differential. The criteria for assessing the efficiency of any latency hiding technique is the extent to which the memory differential can be hidden. This section introduces a new term called β which is an architecture independent measure of the efficiency of a latency hiding technique. β is defined as the average fraction of the memory differential hidden from a memory access and use it to quantify the latency hiding efficiency.

A generic model of a memory hierarchy is shown in figure 5.1. The element χ is used to denote any number of levels and types of memory (e.g. queues, buffers, caches) in the memory hierarchy. Since χ characterises the type of memory system, this will referred to it as a χ memory system. First the following terms are defined for an χ memory system.

- c_M, c_m are, respectively, the execution cost of a main memory and register access.
- δc is the memory differential. It is simply given by $c_M - c_m$.
- $\mathsf{H_r}$, $\mathsf{H_w}$ are, respectively, the cost of a read and write hit in χ.
- $\mathsf{M_r}$, $\mathsf{M_w}$ are, respectively, the cost of a read and write miss in χ.
- $\overline{p}$ is the average perceived access time for the χ memory system.
- δl is the average hidden memory latency. It is given by

$$\delta l = \delta c - (\overline{p} - c_m) = c_M - \overline{p}$$

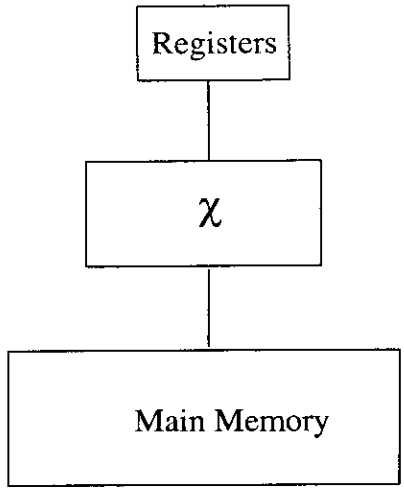


Figure 5.1: χ memory system

- α_r and α_w are respectively, the read and write hit ratio for χ.

The efficiency of a latency hiding technique is the fraction of the memory differential hidden by χ. This is given by :

$$\beta = \frac{\delta l}{\delta c} \tag{5.1}$$

The average perceived access time is given by

$$\begin{aligned} \overline{p} \;=\; & (1-w)[\alpha_r \mathsf{H_r} + (1-\alpha_r)\mathsf{M_r}] \\ & +w[\alpha_w \mathsf{H_w} + (1-\alpha_w)\mathsf{M_w}] \end{aligned} \tag{5.2}$$

where w is the proportion of write accesses. It is known that for any memory system $\mathsf{H_r}, \mathsf{H_w} \geq c_m$ and $\mathsf{M_r}, \mathsf{M_w} \geq c_M$. Substituting into equations 5.1 and 5.2 yields

$$\beta \leq \beta^{ideal} = (1-w)\alpha_r + w\alpha_w \tag{5.3}$$

Where β^{ideal} is used to denote the upper bound on β. When $\alpha_r = \alpha_w$ equation 5.3 reduces to $\beta \leq \alpha$, where α is the hit ratio ($\alpha = \alpha_r = \alpha_w$). This shows that β for any memory hierarchy is bounded by its hit ratio.

By comparing different latency hiding techniques using β, it is possible to predict the miss ratio required by a cache in order to attain the same level of latency hiding through decoupling (see section 5.3).

An alternative and equivalent way of expressing β is $(1-w)\beta^{read} + w\beta^{write}$ where $\beta^{read} \leq \alpha_r$ and $\beta^{write} \leq \alpha_w$. β^{read} and β^{write} are the latency hiding efficiency for, respectively, a read and write access.

Having considered β for a generic memory hierarchy it is now shown how it can be applied to write back and write through single level caches with block size b words.

5.1.1 Write back (WB) cache

In this type of cache a write hit only writes to the cache. On a read or write miss a block is chosen for replacement and if the block has been modified (referred to as a dirty miss) it is copied back to main memory. Since most write back caches operate a write allocate policy, the required block is loaded into the cache on both a write and read miss. The proportion of dirty misses in a program is denoted as w_b. For a write back cache with a fully interleaved memory in which the first word in a block takes MD cycles to arrive; all subsequent words from the block are assumed to take a single cycle.

$$\begin{aligned} \mathsf{H_r} &= c_m \\ \mathsf{H_w} &= c_m \\ \mathsf{M_r} &= (1+w_b)c_M + (1+w_b)(b-1) \\ \mathsf{M_w} &= (1+w_b)c_M + (1+w_b)(b-1) \end{aligned}$$

Therefore, for a write back cache (see Appendix B)

$$\beta = \beta^{ideal} - \frac{(1-\beta^{ideal})}{\delta c}(w_b c_M + (1+w_b)(b-1)) \tag{5.4}$$

Let us consider a cache with $b = 8$ words, $c_M = 61$ cycles, $\delta c = 60$ cycles, $w = 0.12$, $w_b = 0.5$ and $\alpha_r = \alpha_w = 0.9$ then $\beta = 0.83$. This means that on average 83% of δc will be hidden by this write back cache.

5.1.2 Write through (WT) cache

In a write through cache, copies are kept consistent by writing values through to the cache and main memory. Write through caches tend to operate a no write allocate policy so that on a write miss the block is not loaded into cache. For a fully interleaved memory.

$$\begin{aligned} \mathsf{H_r} &= c_m \\ \mathsf{H_w} &= c_m + \delta c \\ \mathsf{M_r} &= c_M + (b-1) \\ \mathsf{M_w} &= c_M \end{aligned}$$

Substituting into the equation (see Appendix B) for β gives

$$\beta = \beta^{ideal} - w\alpha_w - \frac{(1-\alpha_r)(1-w)(b-1)}{\delta c} \tag{5.5}$$

When the same values from the write back example above are inserted into this equation, $\beta = 0.78$. Figure 5.2 shows a 2-dimensional representation of the variation of β with a hit rate (denoted as a percentage) and memory differential. It can be seen that for large values of δc both types of caches saturate as the memory differential becomes the dominant term in equations 5.4 and 5.5. It will also be noticed that at high hit rates the WB cache has a higher β than the WT cache. The situation is however reversed when the hit rate is below 82%.

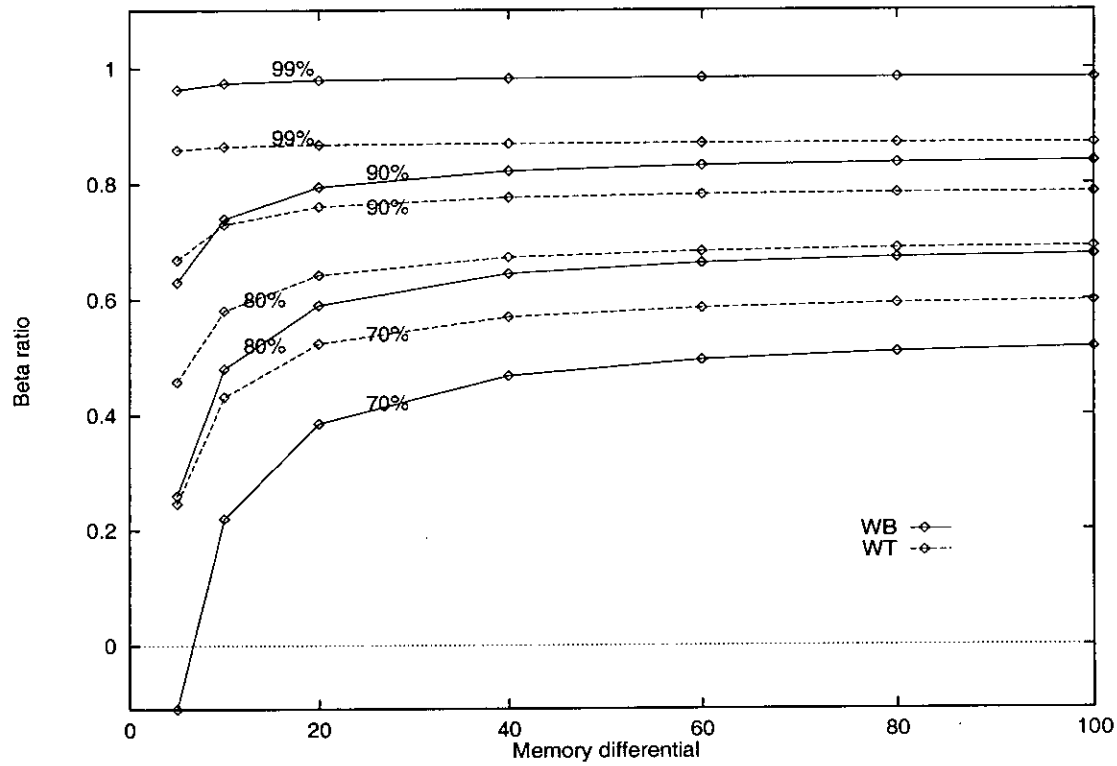


Figure 5.2: β for WT and WB cache, for various values of α

5.1.3 β for the decoupled model

This section describes how β was derived for the decoupled architecture. The semantics of a decoupled store (see Chapter 2) are such that the cost of a store operation is equal to the cost of writing a value to the decoupled memory. Assuming this cost is equal to the time for a register access, $\beta^{write} = 1$.

To derive β^{read} the notion of the decoupling distance is introduced. The decoupling distance is a function of the time the AU sends an address, the DU last

wrote a value, the DU fetches a value and the type of decoupled memory. The decoupling distance for the ith load made via an optimised decoupled memory can be defined using the same notation as in Section 3.1.

$$d_i = \begin{cases} l_i^{DU} - l_i^{AU} & if \quad (l_i^{AU} > s_i^{DU}) \\ \infty & otherwise \end{cases}$$

When the load is made via a basic decoupled memory the decoupling distance is given by

$$d_i = \begin{cases} l_i^{DU} - l_i^{AU} & if \quad (l_i^{AU} > s_i^{DU}) \\ l_i^{DU} - s_i^{DU} - 1 & otherwise \end{cases}$$

The perceived load latency for a decoupled load of i can be defined as

$$pll(d_i) = \begin{cases} c_d & if \quad (d_i \geq c_M) \\ \delta c + c_d & if \quad (d_i \leq 0) \\ \delta c - d_i + c_d & otherwise \end{cases} \tag{5.6}$$

Using equation 5.6 the function pll is bounded between c_d, the time to access the decoupled memory and the maximum load latency cost $\delta c_M + c_d$. The average perceived load latency for a decoupled memory system is trivially given by

$$\overline{pll} = \frac{\sum_{i=1}^{N} pll(d_i)}{N}$$

where N is the total number of decoupled loads in the program. Figure 5.3 illustrates the three different regions of Equation 5.6 when $l_i^{AU} > s_i^{DU}$. The diagram shows the time lines for the AU and DU. The points on the lines depict the times at which operations occur in the execution of a decoupled load. Since the fetch can occur anywhere between s_i^{DU} and o_i^{DU} the AU and DU will be

- coupled in region 1.
- partially decoupled in region 2
- fully decoupled in region 3.

5.1.3.1 Fetch schemes

The last operation in the decoupled load is the fetch instruction issued by the DU. The fetch instruction occupies a single issue slot, tries to read data from the decoupled memory and write it to the DU register file. Since the fetch can be issued at anytime between s_i^{DU} and o_i^{DU} (see figure 5.3) the upper and lower bound of β^{read} can be evaluated by considering two different fetch schemes:

1. The earliest issue fetch (EIF) scheme can only be used with the optimised decoupled memory. In this scheme the DU tries to fetch the data at the earliest possible time. This time is equal to s_i^{DU} plus the time for the store to complete, $s_i^{DU} + 1$. The EIF scheme minimises the decoupling distance and provides worst case results for β^{read}.

2. The latest issue fetch (LIF) scheme. In this scheme the DU issues the fetch at the earliest time the first operation uses the loaded datum less the time to access the decoupled memory. The fetch time is given by $o_i^{DU} - c_d$. The LIF scheme maximises the decoupling distance and provides best case results for β^{read}.

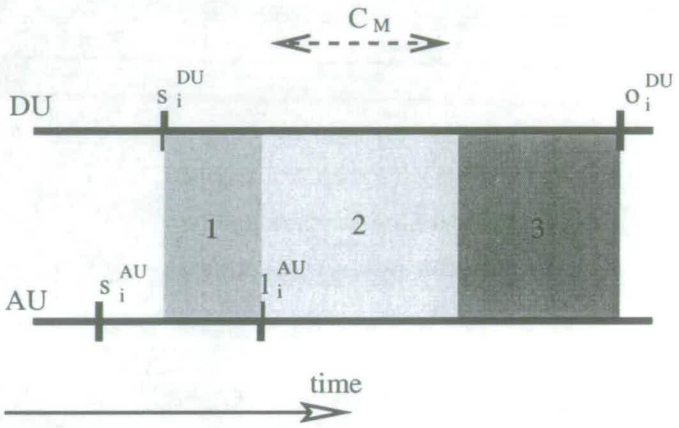


Figure 5.3: Time lines for the ith decoupled load during execution

5.2 Simulation technique

The purpose of the experiments was to find the limitations of access and control decoupling in a 3-stream architecture when ILP is maximised. The experiments simulated an idealised decoupled machine with unlimited resources, idealised out-of-order execution and perfect data dependency analysis. This meant the machine had maximum ILP and optimal slippage. The latter is a consequence of the unlimited addressing resources and the perfect data dependency analysis removing false synchronisation points.

There was no speculative execution but operations from different loop bodies could execute in parallel if control dependencies permitted. Table 5.1 shows the four programs selected from the benchmarks shown in Table 4.2. These were chosen to represent varying degrees of vectorisation and decoupling efficiency (see

Table 4.2). It is recognised that this only represents a small subset of programs but this restriction was felt to be acceptable for the following reasons. Researchers typically only simulate part of a program's total execution time because full simulations can be prohibitive. In this chapter (see Section 5.3.4) it is shown that some programs do need to be simulated in full because of identifiable phases in the program which exhibit differing behaviour. It was therefore decided to perform full simulation of a small number of programs to completion.

Table 5.1 shows the measured total number of operations, the total number of loads, a breakdown by unit of the ILP [1], the fraction of loads (FOL) and the fraction of operations (FOP). The measurements were made with a memory differential of 0 cycles and a floating point cost of 5 cycles.

Program	Measured										
	Ops.	Loads	ILP			FOL(%)			FOP(%)		
	($*10^9$)	($*10^8$)	cu	au	du	cu	au	du	cu	au	du
ARC2D	6.797	13.139	302	1255	2155	8	0.1	91.9	8	34	58
TRFD	1.857	4.419	3	1067	1676	0	0.1	99.9	0.1	38.9	61
FLO52Q	2.244	4.626	30	156	224	7	0	93	7	38	55
QCD2	1.072	1.126	5	7	10	8	2	90	22	33	45

Table 5.1: Characteristics of the benchmark programs

5.3 Experimental results

The three major findings of the experiments into the effectiveness of access decoupling as a latency hiding technique under the pressure of maximum ILP are:

1. access decoupling with an optimised decoupled memory is potentially a powerful latency hiding and bandwidth filtering mechanism.

2. β for access decoupling is comparable to that of a write-back cache with hit ratios between 88% and 99%.

3. access decoupling with a basic decoupled memory can not hide large memory latency, even with optimal slippage.

To provide the most favourable conditions for latency hiding the execution cost of AU loads and address computations were set to 1 and 0 cycles respectively.

[1] These ILP measurements are high but are within the bounds of previous studies [54]

The floating point and CU load latency were 1 cycle. Communication between units carried no cost.

Figure 5.4 shows the variation of relative increase in execution time as a function of the memory differential. The relative increase is given by

$$\frac{T(l) - T(0)}{T(0)}$$

where $T(l)$ and $T(0)$ are the times to execute the programs when the memory differential is l and 0 respectively. It can be seen that, even with optimal slippage, access decoupling using the basic decoupled memory model is sensitive to increases in memory differential. Only TRFD has sufficient parallelism to hide latency upto 200 cycles. For FLO52Q, a memory differential of 80 cycles increases the program execution time by a factor of 12, compared with a zero-cycle memory latency.

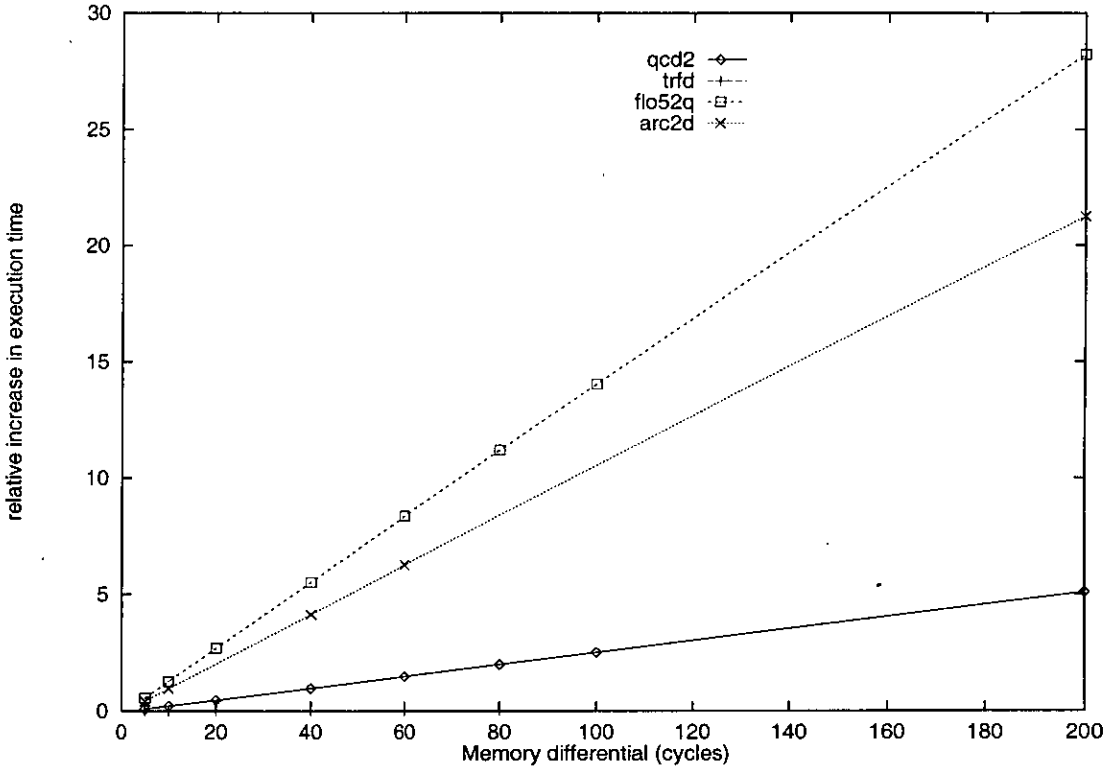


Figure 5.4: Basic decoupled memory model; FP latency = 1

Figure 5.5 shows the effect of introducing the optimised decoupled memory. It can be seen that all four programs show little variation with increases in the memory differential. For FLO52Q, a memory differential of 80 cycles only results in a relative increase of 0.6% in execution time. Figure 5.6 shows that if the floating point latency increases to 5 cycles, effectively slowing down the DU, the sensitivity to memory latency decreases still further. This is due to there being

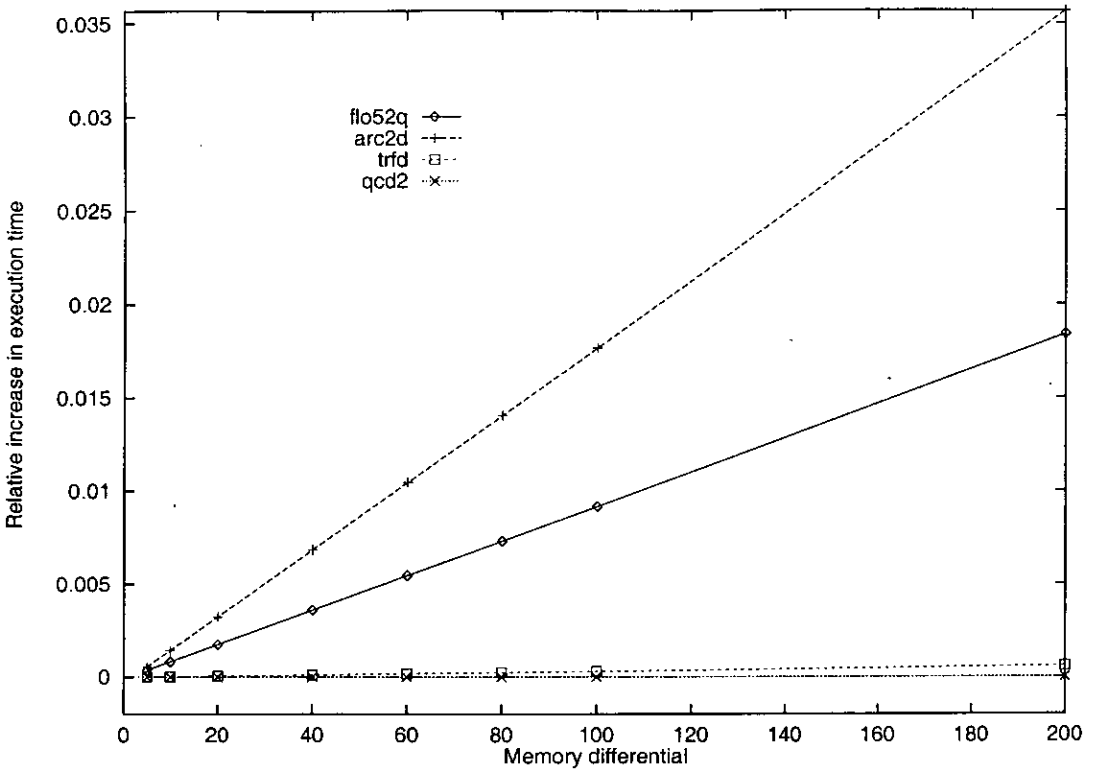


Figure 5.5: Optimised decoupled memory; FP latency = 1 cycle

more opportunity to overlap memory operations and therefore hide more of the memory latency..

Figure 5.7 shows the simulation results for a more realistic case in which AU loads are decoupled and AU address computations take a single cycle. The results show that all the programs still remain insensitive to increases in the memory differential.

One possible criticism of these findings is that the code partitioning has fortuitously biased the experimental results. If most of the loads on a program's critical path were executed on the CU then the experiments would show positive results for access decoupling (recall that CU loads perceive a fixed cost of 1 cycle). To answer this criticism the experiments were repeated with all previous CU loads executed on the DU. DU. Figure 5.8 and 5.9 show that whereas three of the programs are still insensitive to the memory differential the increase in QCD2's execution time is comparable to results for a basic decoupled memory model (see figure 5.4).

The explanation for QCD2's poor results can be found in the routines `LADD`, `LMULT` and `PRANF` where scalar loads previously executed on the CU dominate the critical path. These loads, executed on entry to the routines, can not decouple and perceive all of the memory differential. The three routines are most

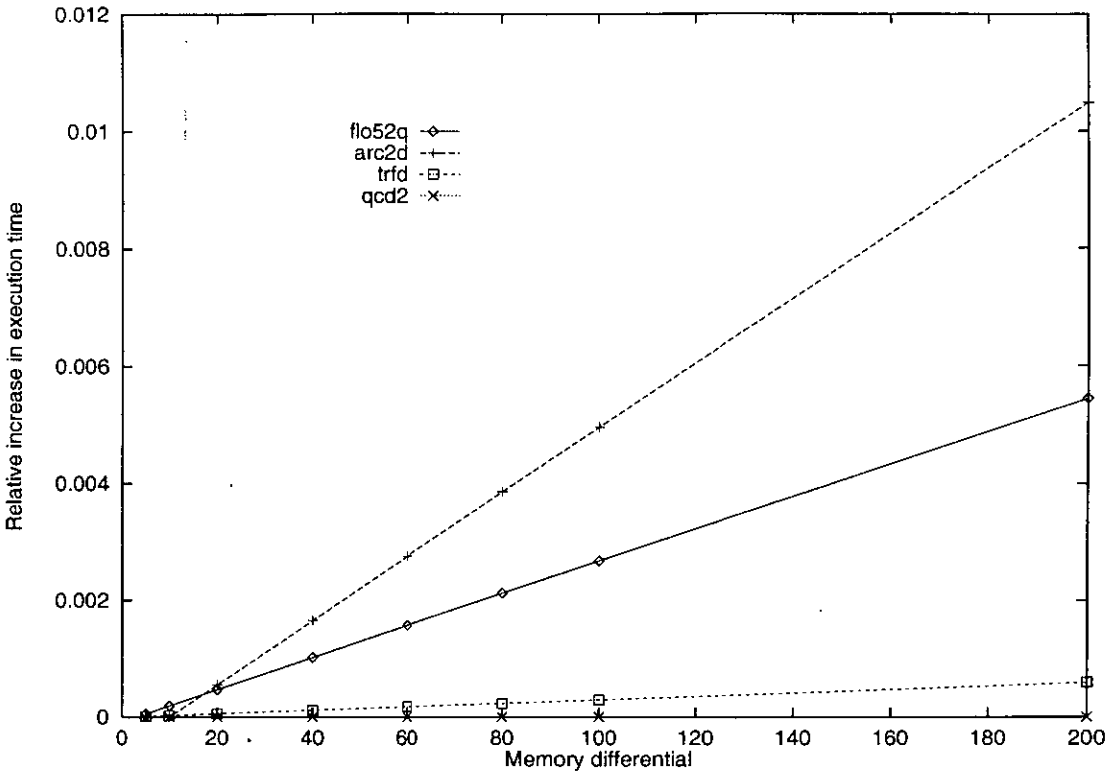


Figure 5.6: Optimised decoupled memory; FP latency=5 cycles

frequently called from inside a while loop in the CHOOS routine. By inlining the three routines it is possible to hoist the loads out of the loop. The effect of this simple optimisation can be seen in figure 5.9. The results show clearly that QCD2 is once again insensitive to increases in the memory differential.

It can concluded that under conditions of maximum ILP, access decoupling with the optimised decoupled memory is still effective at hiding memory latency. It has been shown that higher floating point latencies improve latency hiding and that the results are independent of the partitioning of loads between CU and DU.

5.3.1 Bandwidth filtering

The optimised decoupled memory also acts as a bandwidth filter by reducing the number of accesses that need to go to main memory. This finding is reinforced by the results in Table 5.2 which show the distribution of the decoupled loads in the programs. The table shows the percentage of loads which are coupled(loads perceive the full memory differential), fully decoupled(loads perceive a single cycle latency) and partially decoupled (loads perceive part of the memory differential). The column labelled 'cached' is the percentage of decoupled loads that are cached in the optimised decoupled memory (see Chapter 3). Since cached decoupled loads

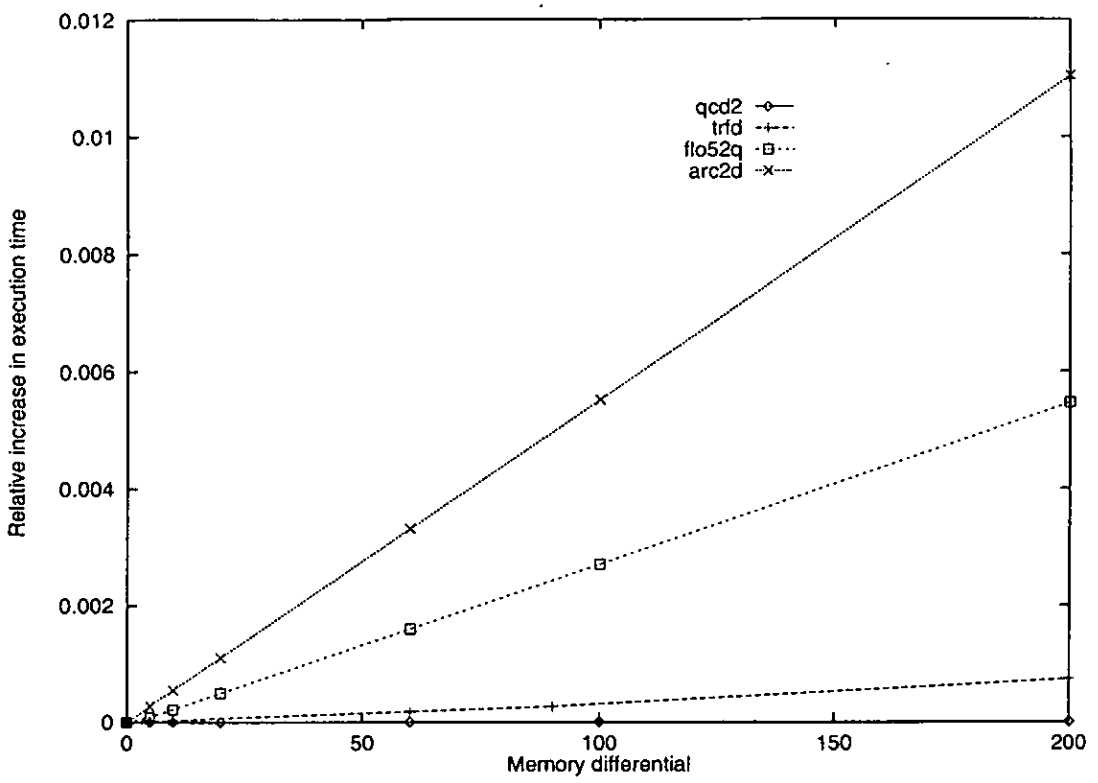


Figure 5.7: AU and DU with decoupled loads; FP latency=5 cycles

will not access main memory this column also shows the bandwidth filtering. It can be seen that, with the exception of TRFD, between 77% and 98% of operations are satisfied in the decoupled memory.

In TRFD almost 50% of loads are either partially decoupled or coupled. This phenomenon is due to induction variables serialising frequently executed loops. These induction variables can be removed using generalised induction variable elimination [23]. The effect of this optimisation can be seen in the row labelled 'optimised TRFD'. The increase in parallelism allows the AU to slip further ahead of the DU, increasing the number of fully decoupled loads from 14.3% to 49.8%.

Clearly the unlimited resources in the decoupled model makes the levels of bandwidth filtering unrealisable. For the four programs the experiments show that the maximum capacity of the decoupled memory is between 65% and 75% of the DU working set size. Although this capacity could not reasonably be implemented, it is still expected that high filtering would occur for more realistic degrees of ILP.

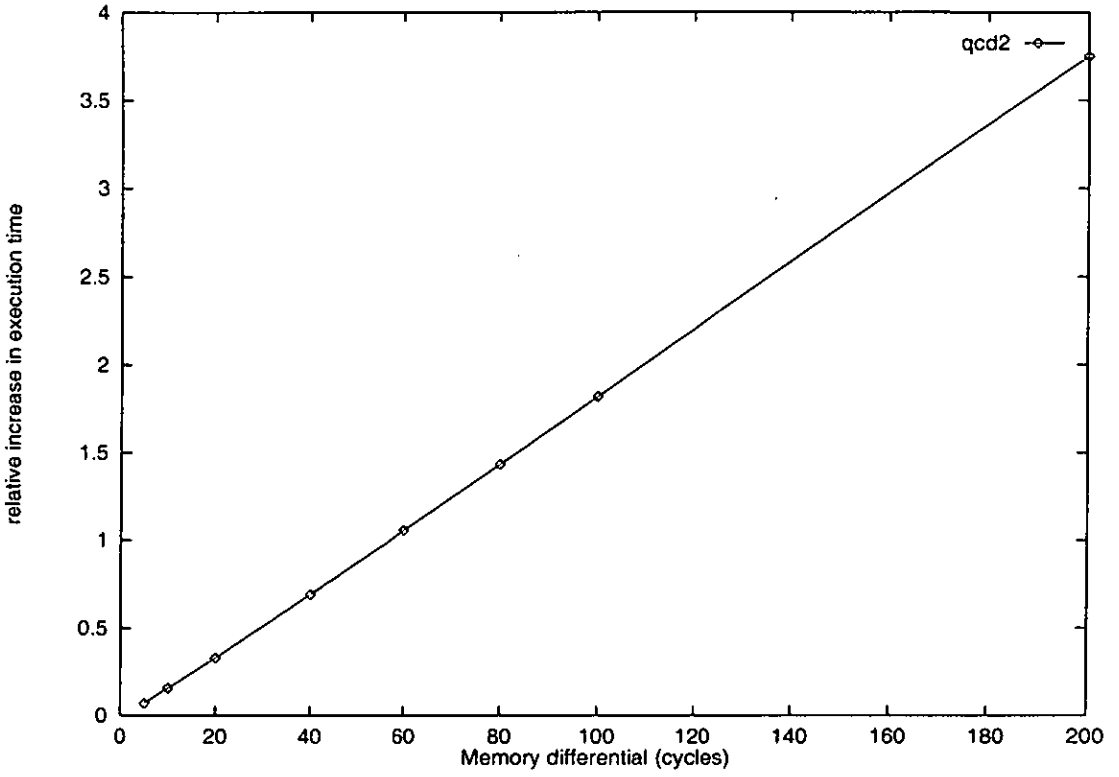


Figure 5.8: Relative increase in execution time with no CU for QCD2

5.3.2 Comparisons of latency hiding efficiency

The motivation for deriving β was to provide a way of comparing the latency hiding efficiency of different techniques. Since caches are the most widely used technique for hiding latency it was decided to compare access decoupling against a write back (WB) cache. A full derivation of β for a write back cache is discussed in Appendix B.

In the experiments β was computed for the decoupled machine by measuring the average perceived load latency. Table 5.3 shows a comparison of β against the WB cache hit rate required to achieve an equivalent degree of latency hiding.

Program	Decoupled load			
	cached (%)	Full (%)	Partial (%)	Coupled (%)
ARC2D	97.5	1.9	0.0	0.6
QCD2	77.3	1.8	0.4	20.5
FLO52Q	89.6	3.5	0.4	6.4
TRFD	35.8	14.3	25.3	24.5
optimised TRFD	49.4	49.8	0.0	0.8

Table 5.2: Decoupled loads for benchmark programs

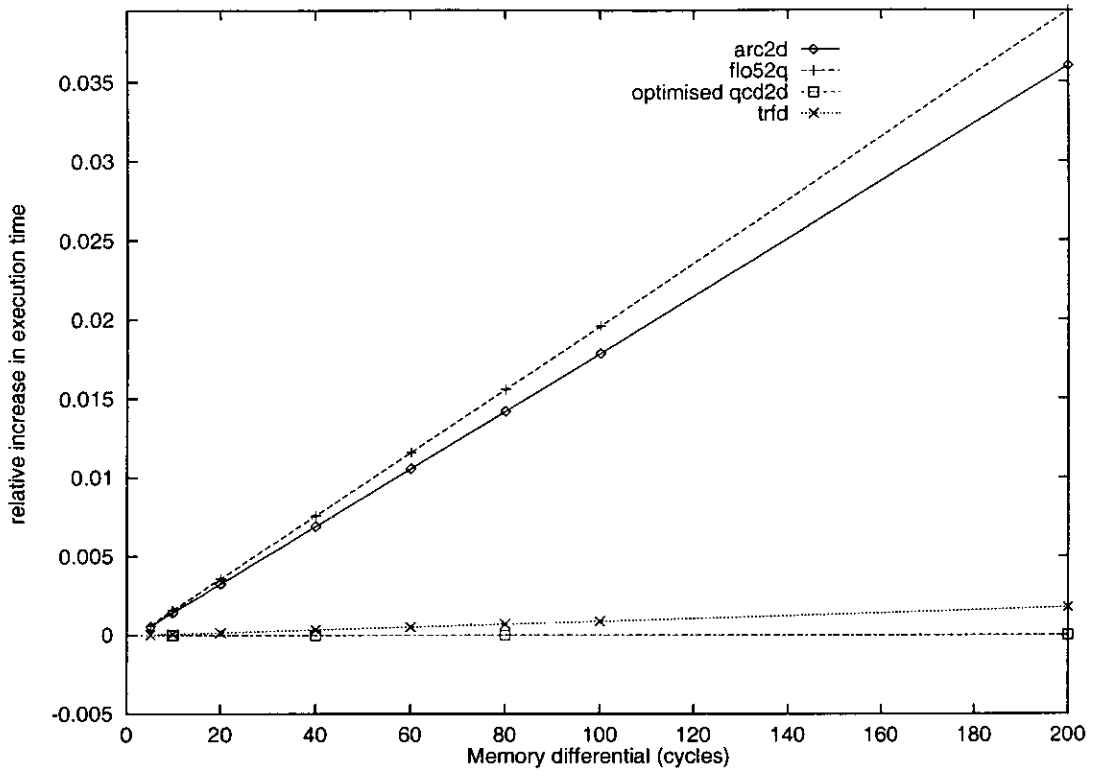


Figure 5.9: Relative increase in execution time with no CU

The term b in columns 3,4 and 6 is used to denote the cache block size. The values in the columns labelled β are for the decoupled machine.

For example, in ARC2D, when the memory differential is 60 cycles β is 0.99. For comparable latency hiding efficiency, a WB cache with block size 8 words would require a hit rate of 99.8%. It is clear that the comparable hit rates are high, in the range 88% to 99%. The table also shows that when the memory differential is reduced to 5 cycles the comparative WB hit rates increase. When the block size is reduced to 2 words the comparative hit rates are similar to those when $\delta c = 60$ and $b = 8$. This observation reflects a well known result [45] that the performance of WB caches degrades when blocks are too large.

program	$\delta c = 5$			$\delta c = 60$	
	β	hit rate		β	hit rate
		$b = 8$	$b = 2$		$b = 8$
ARC2D	0.99	99.8	99.7	0.99	99.8
FLO52Q	0.96	99.1	98.2	0.94	96.6
QCD2	0.82	95.1	90.4	0.82	89.2
TRFD	0.71	92.1	84.6	0.80	88.2
optimised TRFD	0.99	99.7	99.5	0.99	99.6

Table 5.3: β versus WB cache hit rate for different block size

5.3.3 Decoupled fetch schemes

Section 5.1.3.1 discussed the significance of the EIF and LIF schemes. These two schemes, respectively, give the minimum and maximum latency hiding efficiency of any static or dynamic scheduling technique. The measurements of β^{read} in Table 5.4 show that for the optimised decoupled memory the difference between the two schemes is small. The exception is TRFD.

In TRFD the difference is 0.36 and 0.41 for δc of 5 and 60 cycles respectively. This indicates that value lifetimes (measured as the distance in cycles between when a value is defined and used) are significantly greater in TRFD than for the other programs. This finding is reinforced by the results in Table 5.2 which show the distribution, by type, of the decoupled loads in the programs. The table shows the percentage of loads which are coupled, fully decoupled and partially decoupled. The column labelled 'cached' is the percentage of decoupled loads that are cached in the optimised decoupled memory (see section 3.1). The table shows that for TRFD less than 36% of decoupled loads are cached in the decoupled memory. Whilst for ARC2D the figure is 97.5%. These results, plus the data in Table 5.4, lead us to the conclusion that the temporal locality exposed by decoupling in ARC2D, QCD2 and FLO52Q is far higher than in TRFD.

The EIF scheme's poor latency hiding efficiency, in TRFD, is due to the induction variables in the most frequently executed routine `CHOOS`. As described in Section 5.3.1 the induction variables can be removed using generalised induction variable elimination. The effect of this optimisation can be seen in the row labelled 'optimised TRFD' in Table 5.4 and 5.2. The optimisation has increased β^{read} for both schemes and improved the decoupling between the AU and DU, as seen by the larger number of fully decoupled loads

5.3.4 Temporal behaviour of β^{read}

The results for QCD2 in Table 5.4 and figure 5.7 appear to show contradictory results; low values of β^{read} in the former, but insensitivity to memory differential in the latter. To investigate this apparent contradiction β^{read} was sampled at constant intervals throughout the program's simulation. Figures 5.10 and 5.11 show respectively, the interval and cumulative variation of β^{read} with time.

It is noticeable in Figure 5.10 that there are distinctive phases in the program's execution where β^{read} remains almost constant. For approximately 70% of the execution time the latency hiding efficiency is greater than 90%. From figure 5.11 it can be seen that towards the end of the execution there is a sizeable reduction in β^{read}. This is due to a large number of loads perceiving on average 34%

program	fetch scheme	β^{read}	
		$\delta c = 5$	$\delta c = 60$
ARC2D	LIF	0.99	0.99
	EIF	0.98	0.98
FLO52Q	LIF	0.96	0.93
	EIF	0.96	0.90
QCD2	LIF	0.79	0.79
	EIF	0.77	0.77
TRFD	LIF	0.67	0.77
	EIF	0.29	0.36
optimised TRFD	LIF	0.99	0.99
	EIF	0.50	0.49

Table 5.4: β^{read} for EIF and LIF schemes

of the memory differential. Fortunately, most of these loads can be executed concurrently so they have little effect on the critical path. This explains why QCD2 is insensitive to memory differential but has a relatively low value of β^{read}.

The different phases in the graphs justify the argument in section 5.2 that in order to obtain reliable results it is sometimes necessary to perform full program simulations. However such an approach is too expensive in terms of the compute resources required per experiment. To solve the problem of these competing demands a sampled simulation technique was developed. A description of the technique is provided in Appendix C.

5.4 Implications of the experimental findings

Table 5.1 shows that the CU is under-utilised, only executing between 0.1 to 22% and 0 to 8% of operations and loads, respectively. This suggests that that the dedicated hardware support for control decoupling is unnecessary when using the partitioning algorithm of the OCTAVE compiler. In all future experiments the 2-stream decoupled architecture was used with the latest issue fetch scheme.

Table 5.5 shows the characteristics of the benchmark programs for the 2-stream decoupled architecture when using the sampled simulation technique. This table shows the IPC, latency hiding effectiveness (LHE) and β values when architectural resources are unlimited. The columns labelled yes and no denote if the bypass mechanism is included. The IPC values represent the upper bound for the decoupled architecture and indicate the average level of parallelism in the programs.

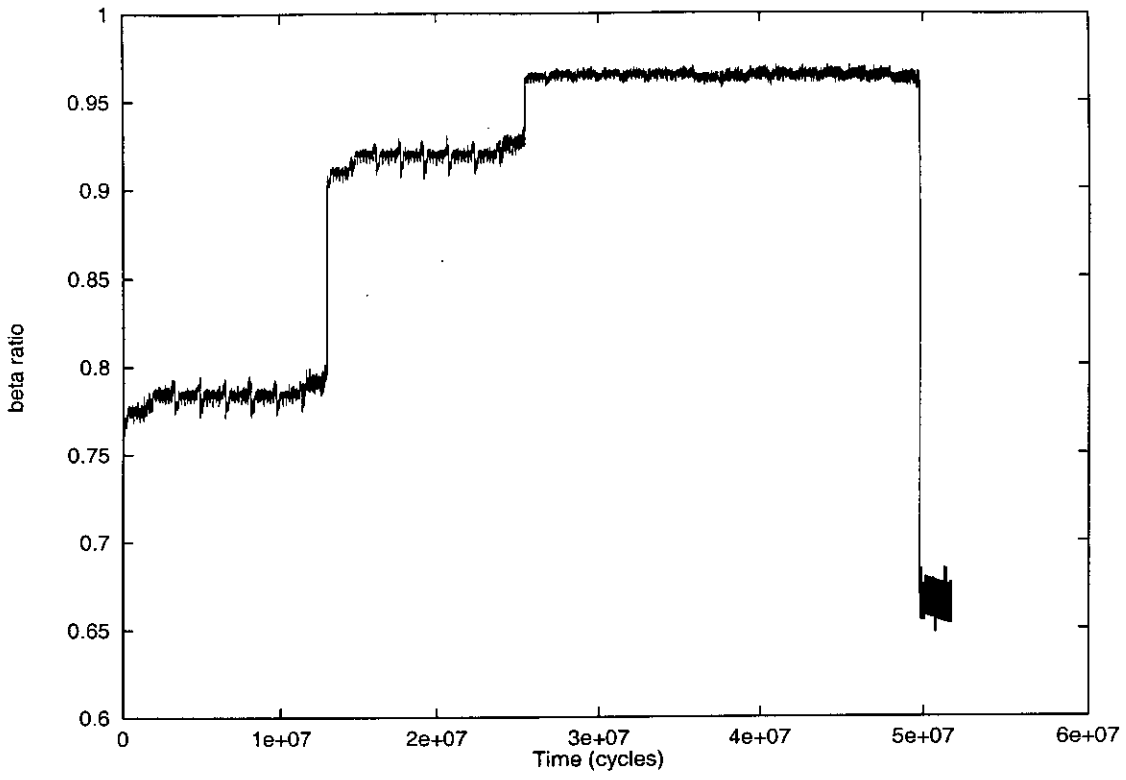


Figure 5.10: QCD2 : interval variation of β^{read}

Table 5.5 shows that for all programs the optimised decoupled memory improves β by between 190% and 380%. The latency hiding effectiveness is seen to improve in some cases by a factor of 6. However, ADM shows that even though β increases by almost 240%, the latency hiding effectiveness remains unchanged. It can be speculated that this is due to the nature of the two measurements. β is a function of the average perceived latency for all loads while the latency hiding effectiveness is only dependent on accesses on the critical path. This finding shows that concentrating on the average access time can be misleading, it is only the loads on the critical path that matter.

It can also be observed that ARC2D has been removed from the benchmark suite. This was due to the program behaving in a similar way to TRFD, high ILP and decoupling well, and long execution times required (approximately 2 hours per data point), even with sampling. Since it contributed little to the spectrum of program behaviour being investigated, it was decided to remove it from the benchmark suite.

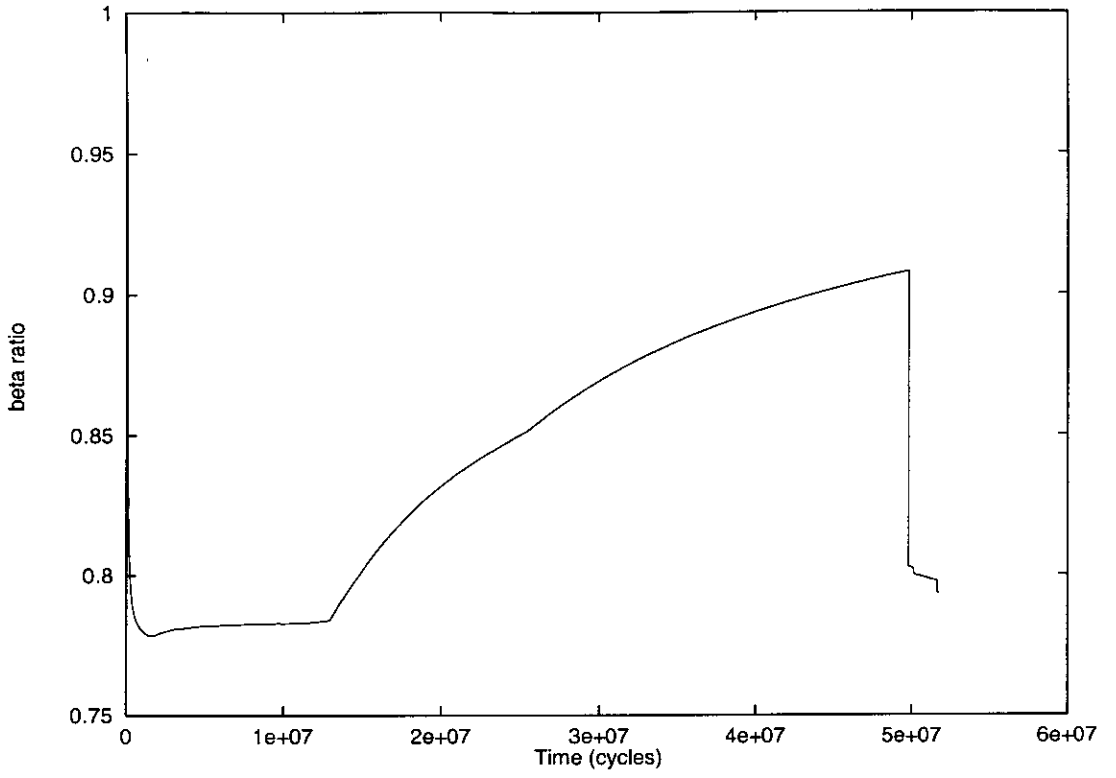


Figure 5.11: QCD2 : cumulative variation of β^{read}

5.5 Summary

This chapter has described a study into the theoretical limits of latency hiding through decoupled execution when ILP is pushed to a maximum. To focus on the limits of decoupling, the machine model had unlimited address computation resources, ideal out-of-order execution and perfect data dependency analysis.

It has been shown that when ILP is a maximum and dependency analysis is perfect, access decoupling with an optimised decoupled memory remains a powerful latency hiding and bandwidth filtering technique. The optimised decoupled memory uses information about future access patterns to cache data close to the AU and DU. However, unlike a typical cache where resident data may not be accessed, the cached data in the decoupled memory will definitely be used. In [70] a similar technique was discussed but no empirical evidence was given to support its use. The SUNDER architecture employed a similar technique through the use of pending store and load queues in the prefetch engine [18].

This study has also shown that access decoupling with a basic decoupled memory model is not capable of hiding large memory latencies. This result can be made more general by observing that there are no resource constraints or speculative execution, and under these conditions decoupling is identical to any

Program	MD=0	MD=60					
	IPC	IPC		LHE (%)		β	
		no	yes	no	yes	no	yes
ADM	587	104	104	18	18	0.31	0.74
DYFESM	24	10	23	44	97	0.17	0.65
FLO52Q	312	65	311	21	99	0.42	0.94
MDG	36	6	36	16	100	0.35	0.84
QCD2	22	6	7	25	30	0.20	0.61
TRACK	2	0.5	0.5	21	22	0.10	0.22
TRFD	619	619	619	100	100	0.5	0.97

Table 5.5: Program characteristics

other data prefetching technique. It can be stated, for the applications considered when ILP is a maximum, that any data prefetching technique without speculative execution can not hide large memory latencies.

This chapter has also shown that there is no advantage to control decoupling using the partitioning algorithm in the OCTAVE compiler. Results have shown that the low utilisation of the control unit means there is little justification for its inclusion in the decoupled architecture. The 3-stream decoupled architecture and control decoupling are therefore not considered in the rest of this thesis.

It can be concluded that if future designers are able to obtain the levels of ILP and dependency analysis outlined in this paper, access decoupling could still be an effective latency hiding technique.

Chapter 6

Determining Balanced Configurations for a Decoupled Architecture

The previous chapter presented a study into the theoretical limits of access decoupling. This chapter extends the previous study by investigating the effect of superscalar instruction issue on the decoupled architecture. The results show how the speedup varies as a function of the issue width and memory latency. It is found that each program's instruction per cycle ratio (IPCR), the ratio of DU instructions per cycle (IPC) to the AU IPC, is constant. This result allows estimates of good AU and DU issue width configurations. These estimates are verified through experimental simulation and a range of issue widths for which the estimates prove valid are identified. IPCR is also found to be a useful indicator of the effectiveness of access decoupling at hiding memory latency. Programs with high/low IPCRs are found to decouple well/poorly and are more/less effective at hiding memory latencies. Finally, for balanced AU and DU issue width configurations, it is shown how the effectiveness of the latency hiding and IPC varies as a function of the issue width.

6.1 Simulation technique

The experiments use the base-line configuration for the 2-stream architecture shown in Table 6.1. The table also shows the range of architectural parameters investigated in this chapter.

The key aspects of the base-line configuration are that the decoupled memory does not include a bypass mechanism; decoupling is therefore the only mechanism used for latency hiding. There is no speculative execution; instead the cost of loop closing branches is assumed to have been removed by optimisations like

loop unrolling and branch prediction. Data dependency analysis is perfect and false dependencies are removed by renaming. The purpose of examining such an ideal case is to provide the best opportunity for prefetching data, to have high instruction level parallelism (ILP) and to place the greatest pressure on the latency hiding mechanism.

Architecture Components	Base-line Values	Other Values Considered
Floating operation latency (cycles)	5	
Integer/Address op. latency (cycles)	1	
Function units	∞	{DU,AU} = {1,3,5,7,9}
Issue width	∞	{DU,AU} = {1,3,5,7,9}
Register file	∞	
Window size	∞	
Comm. bus latency	0	
Comm. bus width	∞	
Decoupled memory	∞	
Memory bandwidth	∞	
Decoupled memory access latency (cycles)	1	
Bypass mechanism	No	Yes
Memory differential (cycles)	60	
Dependency analysis	perfect	
Renaming	yes	
Code partition	static	

Table 6.1: Configurations for the 2-stream decoupled architecture

6.2 Experimental results

This section describes the experimental findings into the effect of issue width and memory latency on normalised speedup; defined as the speedup relative to a 2-stream decoupled architecture with 1-way issue for the AU and DU. Simulations were performed for all the programs in Table 5.5.

Figures 6.1 and 6.3 show the speedup for ADM for the cases when the memory differential is 0 cycles[1] and 60 cycles[2], respectively. The graphs show how the speedup varies as a function of the AU and DU issue width values. From the similarity of the two graphs it can be inferred that for programs with high ILP access decoupling can completely hide the memory differential. The graphs also show that there is some point, always in the region where the AU issue width

[1]An MD of 0 cycles is used to determine an upper bound on machine performance.

[2]An MD of 60 was chosen because it is comparable to the cost of a second level cache miss (e.g the Pentium Pro has a 50 cycle L2 miss latency[8]) and it assumes a memory system capable of capturing no locality. In practice for a high performance architecture the memory system will be able to reduce the average access time by using first and second level caches.

is less than the DU issue width, beyond which additional AU issue slots provide only a small increase in performance. Figures 6.1 and 6.3 show that for ADM it is possible to find an AU issue width which gives a normalised speedup equal to the number of DU issue slots. For a DU issue width of 9, this point is reached once the AU issue width is 7. This can be interpreted to mean that it takes 7 AU issue slots to provide sufficient data to keep a DU, with an issue width of 9, fully utilised.

Figures 6.2 and 6.4 show the speedup for QCD2 when the memory differential is 0 and 60 cycles, respectively. Figure 6.2 shows that even when MD is 0, there is insufficient parallelism in the program to utilise large AU and DU issue widths; the normalised speedup is only equal to the number of DU issue slots in the region between 1 and 5 slots. Figure 6.4 shows the performance impact of the large 60 cycle memory latency on a program with limited ILP. The graph shows that for QCD2 none of the issue width configurations are able to achieve a speedup greater than 2.5. Figure 6.4 clearly show that there is little performance to be gained by increasing the AU and DU issue widths above 3 instruction slots.

Program Name	IPCR	
	average	σ^n
ADM	1.323	-
DYFESM	1.428	-
FLO52Q	1.343	± 0.007
MDG	1.222	± 0.002
QCD2	1.057	± 0.002
TRACK	0.873	-
TRFD	1.546	-

Table 6.2: Program IPCR

6.2.1 Estimating good issue width configurations

This section shows how the IPCR can be used to estimate good AU and DU issue width configurations and how these estimates can be experimentally verified to be optimal for a range of DU issue widths

Table 6.2 shows the measured IPCR and the variance (σ^n) for all the programs. This ratio has been found to be a program constant and is useful for estimating balanced AU and DU issue width configurations. It can be seen that most of the programs have an IPCR greater than 1; this indicates that the AU issue width would be under utilised in an architecture which had equal AU and DU issue

widths. It can be deduced that possible good configurations of AU and DU issue width would be (1,1), (2,3), (4,5), (5,7) and (7,9) [3].

To find the configuration which ensured the maximum usage of the issue width the total utilisation was chosen as the objective function. The total utilisation is defined as the sum of the utilisation functions for the AU and DU; where the utilisation function is the ratio of the IPC to the issue width. For a DU issue width of 5 Figure 6.5 shows the total utilisation for different AU issue widths. It can be seen that all programs, excluding TRACK, behave in a similar manner. There are identifiable peaks in the graphs corresponding to an AU issue width of 3 and 4 slots. The former peak is caused by the graphs for DYFESM and TRFD which have large IPCRs of 1.43 and 1.55, respectively. It would therefore be expected that the number of AU instruction slots required to keep the DU busy would be lower.

Figure 6.6 shows the average utilisation for various AU and DU issue widths. The graphs show identifiable peaks at AU issue widths of 1, 2, 4 and 5 for DU issue widths of 1, 3, 5 and 7 respectively. For a DU issue width of 9 the optimal AU issue width is less obvious with 6 and 7 giving equal total utilisation values of 1.55. This effect is due to the lack of parallelism in MDG, QCD2 and DYFESM. At large issue widths the law of diminishing return begins to apply. This causes the utilisation function to flatten and makes the optimal issue width configuration harder to determine. It can be concluded therefore that the IPCR is useful for pinpointing good configurations when the DU issue width is in the range 1 to 7. However, for large issue width configurations it serves as a useful initial point to explore the search space. Figure 6.6 also shows that the peak utilisation reduces from 1.7 to 1.5 when the DU issue width is 3 and 9, respectively. This result is a manifestation of the law of diminishing returns, as issue widths increase the utilisation of the available issue width decreases.

6.2.2 IPCR and latency hiding effectiveness

Using the optimal configurations of AU and DU issue widths measurements, this section quantifies the latency hiding effectiveness of decoupling and deduces a relationship between the IPCR and the effectiveness of access decoupling.

Figure 6.7 shows for each program how the latency hiding effectiveness [4] varies

[3]The first and second terms in the brackets are the AU and DU issue width respectively. This estimate is made by finding the ratio of DU and AU issue width which are greater than or approximately equal to the average IPCR of 1.26.

[4]The latency hiding effectiveness is defined as $E = T_{perfect}/T_{actual}$ where T_{actual} is the actual execution time for the decoupled machine. $T_{perfect}$ is the execution time with a perfect

as a function of the different issue width configurations. It can be seen for large issue widths that access decoupling can almost completely hide a memory latency of 60 cycles for ADM, TRFD and FLO52Q. However, for MDG, QCD2 and TRACK, access decoupling is shown to be very poor at hiding latency for large issue widths. DYFESM can be seen to fall between these two groups of programs.

Figure 6.7 shows that some programs experience a deterioration in the latency hiding effectiveness as the issue widths increase. This occurs because although more memory accesses can be executed in parallel, the DU is also able to consume data at a faster rate; this means that more independent operations are required to hide the memory latency.

It can also be observed from Figure 6.7 that for an issue width configuration of (2,3) the system is at least 80% as efficient as a perfect latency hiding mechanism. This result shows that at low issue width configuration access decoupling is reasonably effective at hiding large memory latencies.

Examining Figure 6.7 and Table 6.2 it can be observed that those programs with an IPCR greater than 1.33 are more effective at hiding latency. Whilst QCD2, MDG and TRACK have a lower IPCR and lower latency hiding effectiveness for issue width configurations larger than (4,5). DYFESM is a slight exception to this finding since it has a large IPCR, but as issue widths increase its latency hiding effectiveness reduces faster than programs like ADM [5].

It can be deduced therefore that there appears to be a relationship between the IPCR and latency hiding effectiveness; programs with high/low IPCR appear to be more/less effective at hiding memory latency. This is an intuitive result, if we perceive the IPCR as the number of AU issues slots required to keep the DU issue slots supplied with data. When the IPCR is large the delay between data fetches will on average be larger than if the IPCR is small. This provides greater opportunities for overlapping memory operations and therefore improves the latency hiding effectiveness of decoupling.

Table 6.3 shows the IPC and latency hiding effectiveness of each of the programs. The columns labelled no and yes denote if the optimised decoupled memory is included. The major observations are that, with unlimited reordering and restricted issue width, access decoupling can achieve high ILP. In the case of the configuration (7,9), 5 of the programs achieve average IPC values greater than 10

latency hiding in which each memory access perceives a single cycle latency.

[5] This is due to there being less ILP in DYFESM which causes access decoupling to be less effective at hiding memory latency when the DU issue width is 9. However even with a large issue width configuration of (7,9) DYFESM is still 78% as effective as a perfect latency hiding mechanism.

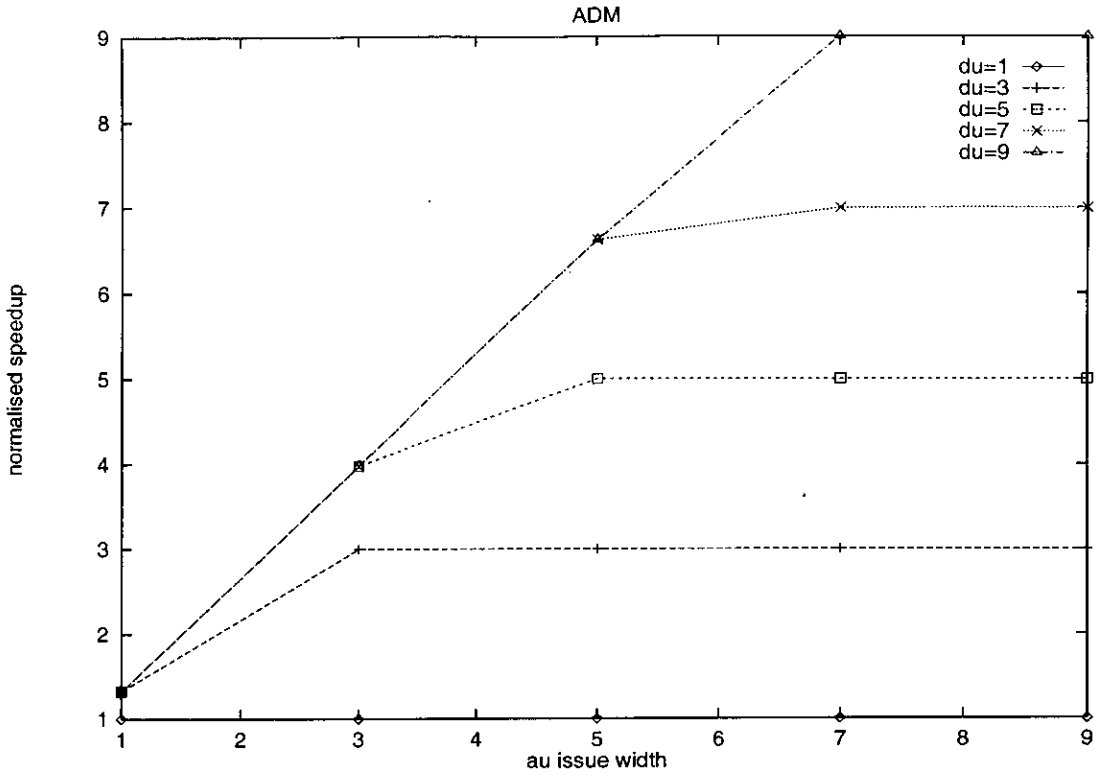


Figure 6.1: ADM normalised speedup for MD=0

when the optimised decoupled memory is used. The latency hiding effectiveness is also seen to be greater than 93% for 5 of the programs. This shows that access decoupling can achieve high IPC and latency tolerance for large memory latencies even at large issue widths. Even in the case of the basic decoupled memory the latency hiding effectiveness is greater than 72% for 4 of the programs. This is in contrast to Chapter 5 which showed decoupling needs to be combined with an optimised decoupled memory to be effective at tolerating latency.

Table 6.3 shows that there are two exceptions QCD2 and TRACK. At small issue width configuration such as (2,3), QCD2 can achieve an IPC of 3.4 and a latency hiding effectiveness of 84%. However, at larger issue width configurations, the latency hiding effectiveness degrades rapidly.

When the MD is 60 cycles, Table 6.3 shows in 6 of the 7 programs the optimised decoupled memory has a higher IPC than the basic decoupled memory. The one exception is ADM which at large issue width configurations is less tolerant to memory latency (see the column labelled LHE in Table 6.3). The reason for this was found by monitoring the execution of the program on an operation by operations basis. It was found that the optimised memory did reduce the average access times [6], but that this allowed non-critical operations to be scheduled earlier,

[6]Table 5.5 shows that β increased from 0.31 to 0.74 when the decoupled memory was opti-

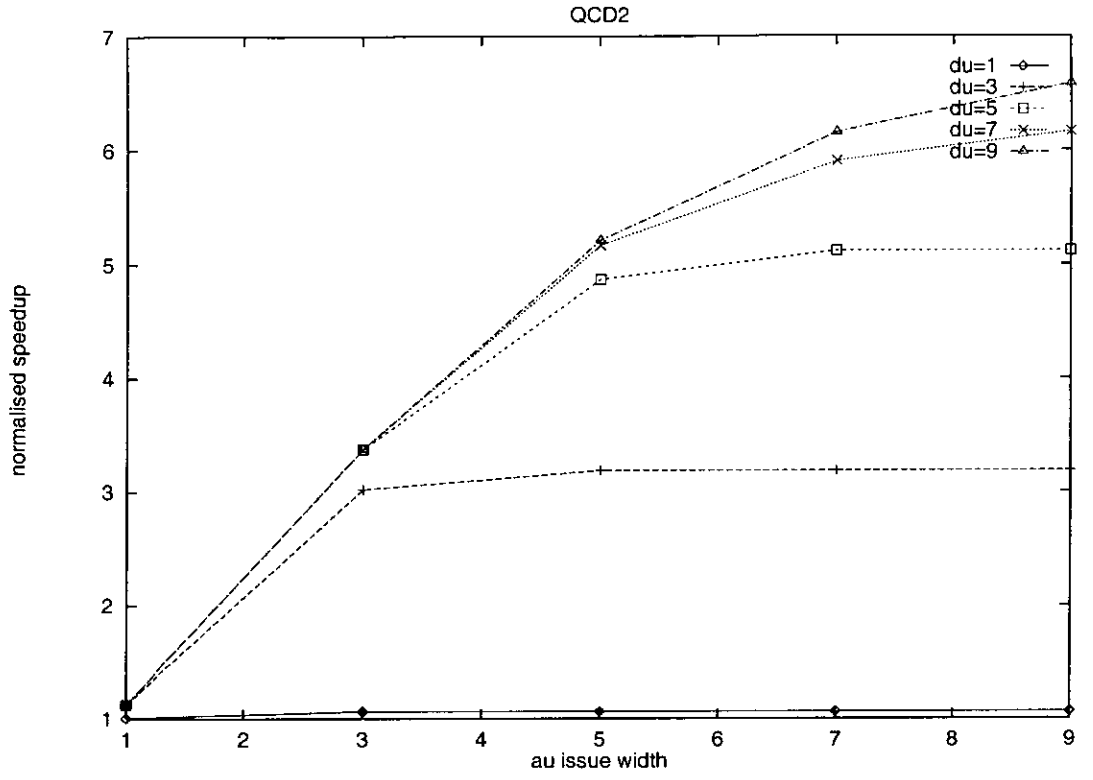


Figure 6.2: QCD2 normalised speedup for MD=0

delaying the execution of operations on the critical path. This result is supported by the findings in Table 5.5 that showed loads on the critical path for ADM did not benefit from the optimised decoupled memory.

A comparison of Tables 5.5 and 6.3 shows that, as expected, the latency hiding effectiveness improves with more realistic issue widths. This is due to the resource constraints forcing data operations to overlap and hide the long memory latencies. This effect is most pronounced in ADM, where the large amount of ILP (see Table 5.1) improves the latency hiding effectiveness from 18% to 97% when the issue width is limited to (7,9).

6.3 Summary

This chapter has explored the relationship between instruction issue width in a decoupled architecture and the resulting latency hiding efficiency. In a decoupled architecture the AU can be thought of as a prefetching engine which feeds the DU with operands. Consequently, balanced rates of execution between these units are required for maximum throughput. The notion of IPCR effectively defines a balanced ratio of issue widths for optimum total utilisation.

mised

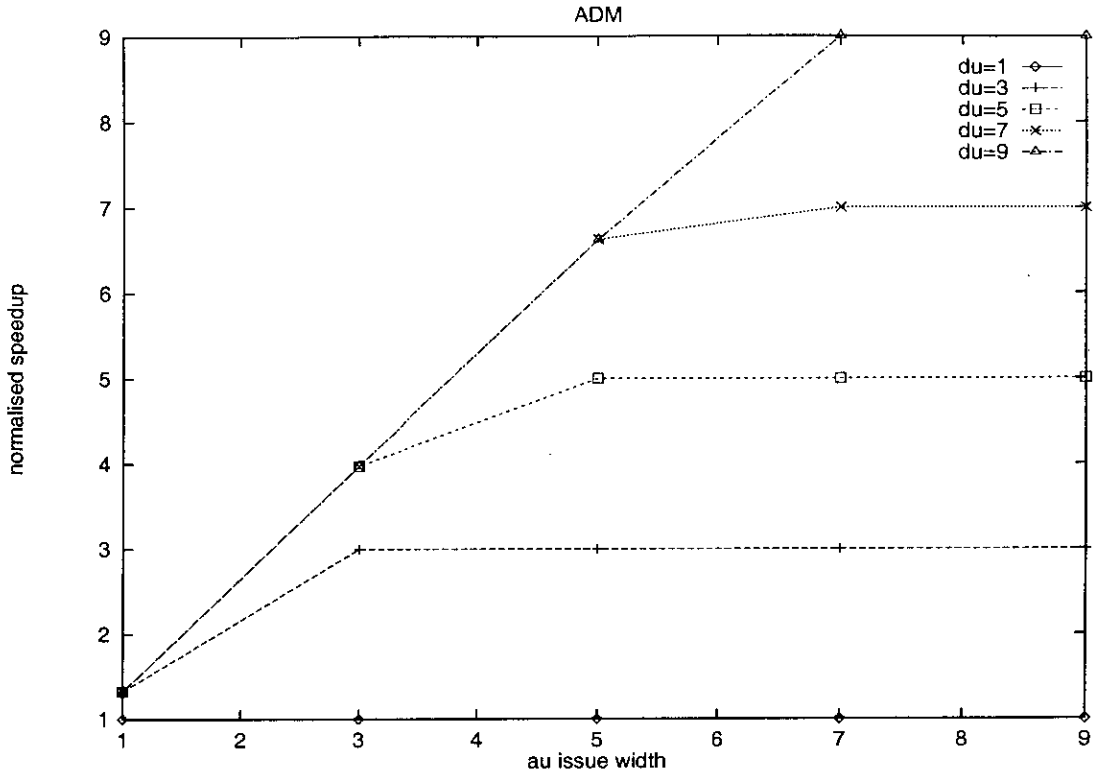


Figure 6.3: ADM normalised speedup for MD=60 cycles

As with all prefetching techniques, the time between the initiation of a memory fetch and the use of its data defines the maximum physical latency that can be hidden. As issue widths increase, the available prefetch time for a given program will diminish. The results confirm that programs with lesser degrees of parallelism see a deterioration in latency hiding effectiveness as instruction issue widths increase.

It has also been shown that programs with high degrees of parallelism are able to completely overlap computation with memory accesses, even for issue widths up to (7,9). This result has been shown to apply even in the case where the optimised decoupled memory is not used. This is in contrast to the results from Chapter 5 where is was shown, when ILP is at a maximum, decoupling alone can not effectively hide memory latency. When issue widths are restricted and reordering unlimited, highly and moderately parallel programs can tolerate large latencies. However as the issue constraints are relaxed we move into the realm modelled in Chapter 5 and decoupling is no longer as effective; at this point the benefits of the optimised memory become more prominent. The promising aspect of this result is that decoupling is still effective in many of the programs even at very high issue widths of (7,9).

It is worth noting that there was no speculative execution in the simulations

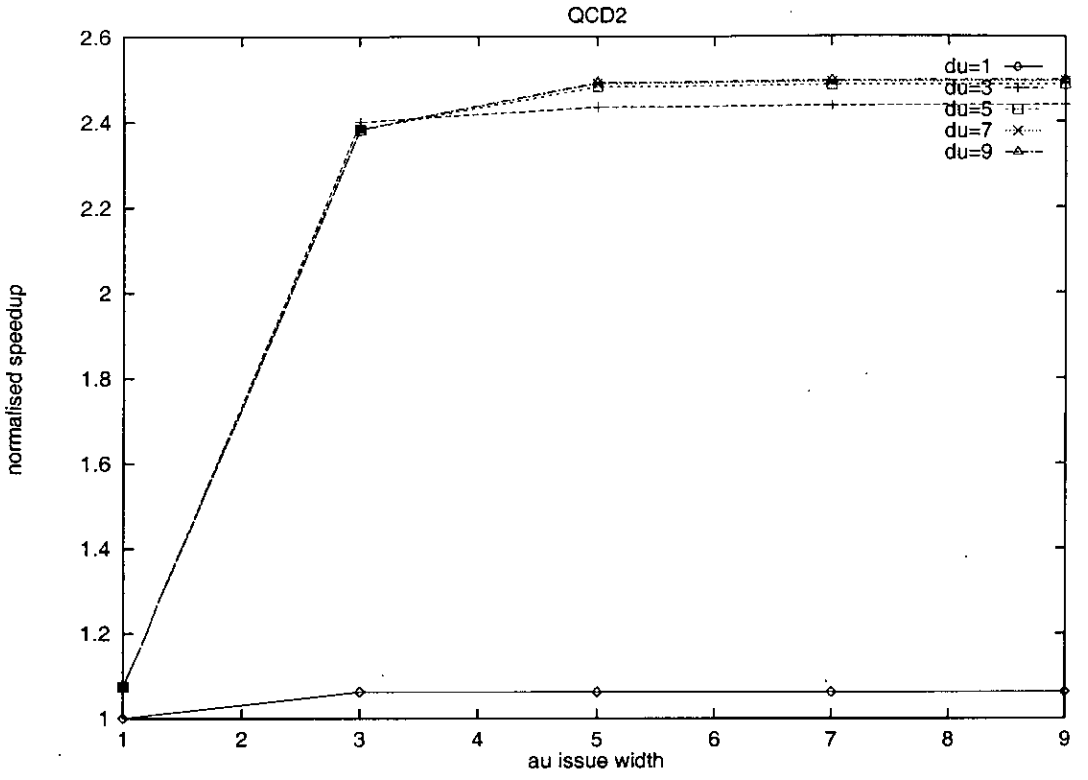


Figure 6.4: QCD2 normalised speedup for MD=60 cycles

presented in this chapter; all control dependencies except loop closing branches are fully resolved before computation continues. This restricts the ability of the system to decouple memory accesses, as well as constraining the amount of ILP that can be exploited.

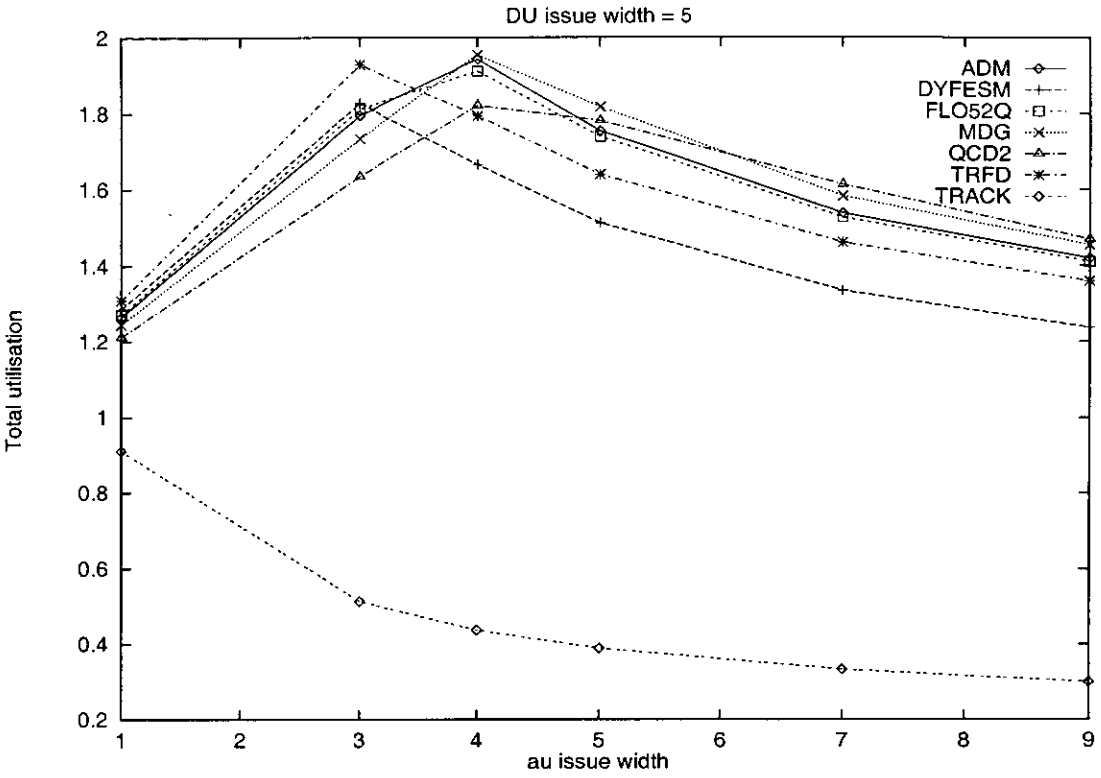


Figure 6.5: Issue width utilisation when DU IW = 5

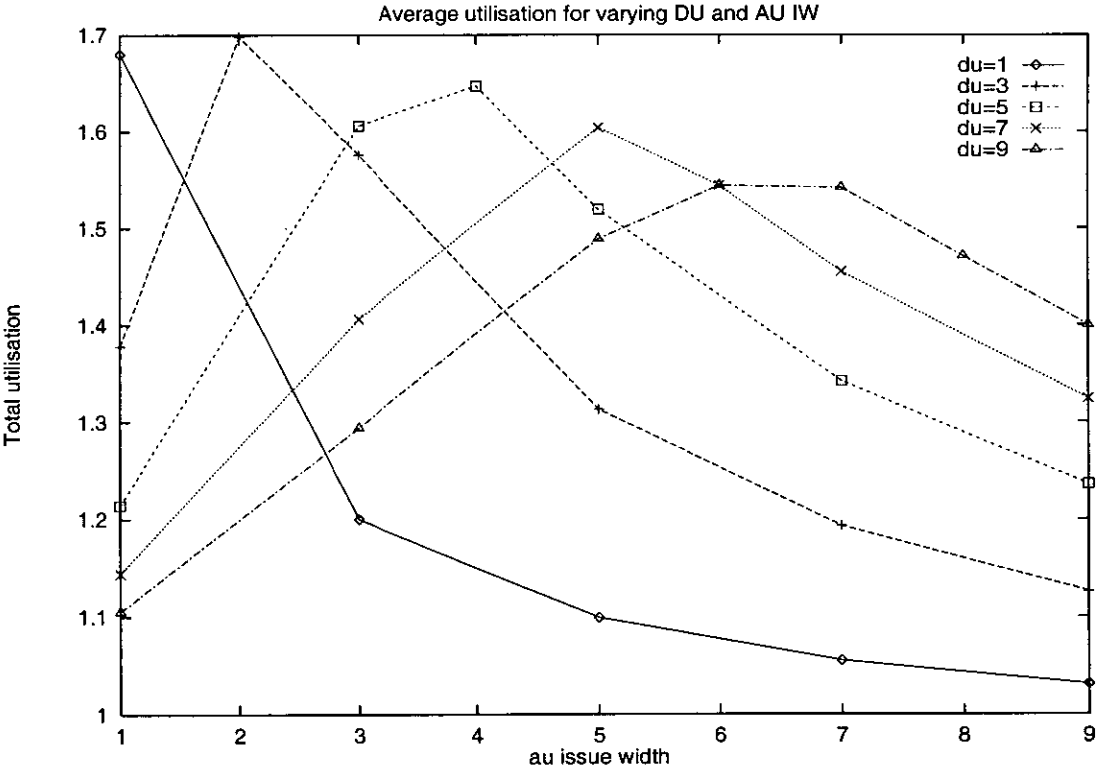


Figure 6.6: Average IW utilisation

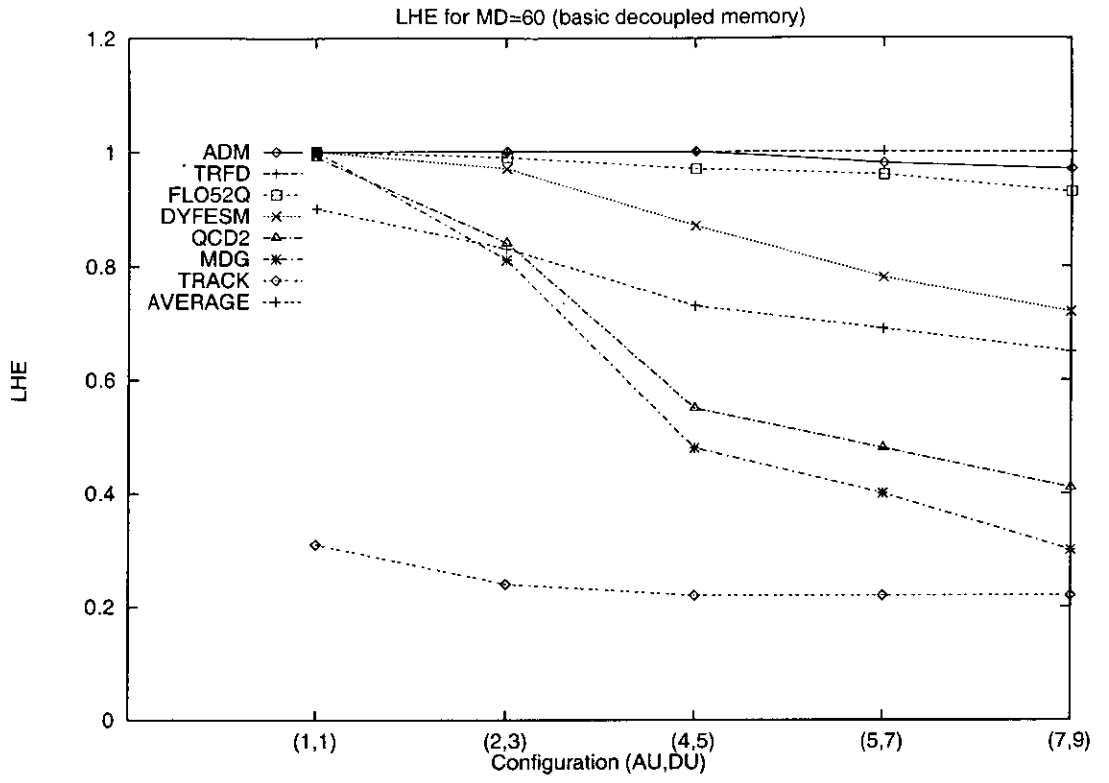


Figure 6.7: Latency hiding effectiveness for the basic decoupled memory (MD=60 cycles)

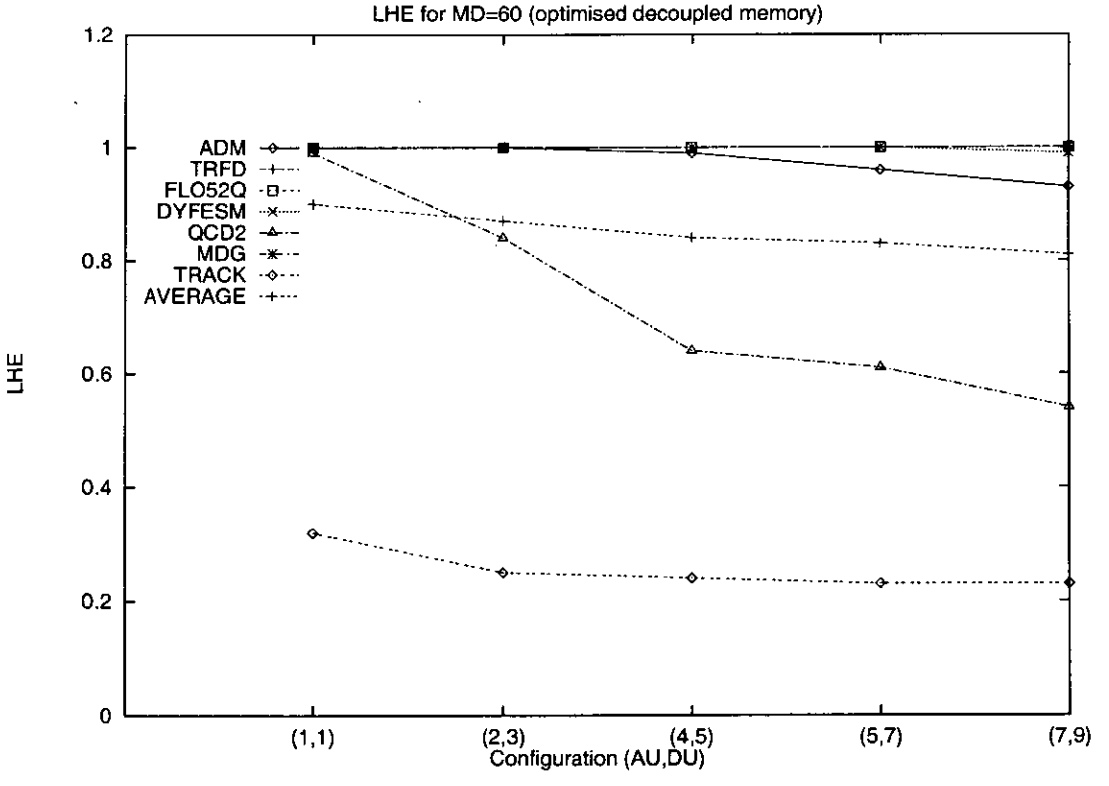


Figure 6.8: Latency hiding effectiveness for the optimised decoupled memory (MD=60 cycles)

Program	Config. (AU,DU)	MD=0 IPC	MD=60			
			IPC		LHE (%)	
			no	yes	no	yes
ADM	(1,1)	1.7	1.7	1.7	100	100
	(2,3)	4.7	4.7	4.7	100	100
	(4,5)	8.6	8.6	8.5	100	99
	(5,7)	11.6	11.4	11.2	98	96
	(7,9)	15.3	14.8	14.2	97	93
DYFESM	(1,1)	1.7	1.7	1.7	100	100
	(2,3)	4.7	4.5	4.7	97	100
	(4,5)	7.6	6.6	7.5	87	100
	(5,7)	9.8	7.6	9.8	78	100
	(7,9)	11.6	8.3	11.5	72	99
FLO52Q	(1,1)	1.7	1.7	1.7	100	100
	(2,3)	4.7	4.6	4.7	99	100
	(4,5)	8.6	8.4	8.6	97	100
	(5,7)	11.6	11.1	11.6	96	100
	(7,9)	15.6	14.5	15.6	93	100
MDG	(1,1)	1.8	1.8	1.8	100	100
	(2,3)	4.4	3.6	4.4	81	100
	(4,5)	8.8	4.2	8.7	48	100
	(5,7)	11.1	4.4	11.1	40	100
	(7,9)	15.5	4.6	15.5	30	100
QCD2	(1,1)	1.8	1.8	1.8	100	100
	(2,3)	4.1	3.4	3.4	84	84
	(4,5)	8.1	4.5	5.2	55	64
	(5,7)	9.4	4.6	5.8	48	61
	(7,9)	11.3	4.6	6.1	41	54
TRACK	(1,1)	1.3	0.4	0.4	31	32
	(2,3)	1.8	0.4	0.4	24	25
	(4,5)	1.9	0.4	0.5	22	24
	(5,7)	2.0	0.4	0.5	22	23
	(7,9)	2.0	0.4	0.5	22	23
TRFD	(1,1)	1.6	1.6	1.6	100	100
	(2,3)	4.9	4.9	4.9	100	100
	(4,5)	8.2	8.2	8.2	100	100
	(5,7)	11.4	11.4	11.4	100	100
	(7,9)	14.7	14.7	14.7	100	100

Table 6.3: IPC and LHE for best configurations of AU and DU with and without the optimised decoupled memory

Chapter 7

Design Issues for a Decoupled Architecture

Out-of-order superscalar architectures try to maximise ILP by reordering operations within the instruction stream. Software and hardware optimisations can improve the degree of reordering by making more operations visible to the issuing hardware.

This thesis is the first known study into a decoupled architecture with out-of-order superscalar units. In the dual stream system, the degree to which operations can reorder within, and between each of the instruction streams will determine the systems overall performance. It is therefore necessary to investigate the dual stream system in relation to hardware and software design issues that determine the degree of reordering. This chapter identifies those issues, and through a series of simulation studies quantifies their effect on IPC and latency hiding effectiveness.

Section 7.1 identifies those design issues critical to reordering in the dual stream system and Section 7.3 explores their effect on the latency hiding effectiveness and scalability of access decoupling. The scalability of decoupling is defined as the variation of latency hiding effectiveness with issue width. One of this chapters findings is that out-of-order memory accesses are critical for hiding large memory latencies. Section 7.4 describes a mechanism to support memory reordering in a dual stream system.

7.1 Design issues

In Section 1.1 it was argued that the degree of reordering was the critical dimension for ILP in an out-of-order superscalar machine. It went on to discuss the reordering factors that determine the degree of reordering. Section 3.4, then

listed those design issues that determine the reordering factors; these issues were referred to as the degrees of freedom for reordering.

This section discusses in more detail the degrees of freedom for reordering in a dual stream architecture. An holistic view is taken of these design issues; with the boundary between hardware and software becoming blurred [32], it is important to consider both hardware and software when investigating the design space of a new machine. As per Table 3.1, the degrees of freedom for reordering are:

1. **Parallelising Optimisations**: Branch operations cause changes in control flow and will, in the absence of parallelising optimisations, limit operation reordering. Branches occur because of procedure calls and returns, loop closing branches and conditional statements. Parallelising optimisations can be supported by hardware e.g. speculative execution, or by software. Software optimisations [63, 4, 26] such as loop unrolling, loop transformations and inlining can reduce the effect of branches, make more operations visible to the reordering hardware and improve machine performance. However, this compiler technology is prohibitively expensive to develop in the course of a thesis; though it is expected that a high performance superscalar architecture would incorporate this state-of-the-art compiler technology. To consider the effect of different levels of compiler optimisations the notion of the reordering scope was developed. Barriers were inserted into the code to restrict the amount of reordering that can occur within an instruction stream; operations were not allowed to reorder across these barriers. The distance between two barriers is referred to as the reordering scope. Increases in reordering scope represents greater sophistication in compiler technology. Conceptually the scope can be thought of as software window for reordering operations. Chapter 8 discusses this notion in comparison with the hardware instruction window. The seven ranges of scope considered in this chapter are:

   RS0 : No out of order execution.

   RS1 : Operations can reorder between different iterations of leaf level loops but not across procedure call boundaries. Operations outside leaf level loops can reorder within basic blocks.

   RS2 : Operations can reorder between different iterations of leaf level loops. Operations outside leaf level loops can reorder within basic blocks.

   RS3 : Operations can reorder between different iterations of nested loops but not across procedure boundaries. Operations outside nested loops

can reorder within basic blocks.

RS4 : Operations can reorder between different iterations of nested loops. Operations outside nested loops can reorder within basic blocks.

RS5 : Operations can reorder within the same procedure.

RS6 : No restrictions on reordering.

2. **Memory ordering**. The sequence in which memory accesses are sent to memory effects the amount of reordering that can take place between memory references. Compiler technology can detect most memory dependencies, but it has been found that in some scientific applications a small number of array references can not be resolved because of complex array subscripts [69]. These unresolved dependencies have been shown to cause a significant reduction in performance. For this reason hardware is required to disambiguate references at run-time. The complexity of this hardware is dependent on the memory ordering scheme supported by the architecture. The three memory ordering schemes considered are :-

   (a) Strong Ordering (STO) : Loads and Stores occur in program order so there is no reordering of memory operations. This is the simplest case requiring no additional hardware or software complexity.

   (b) Semi Strong Ordering (SSO): Loads/stores preserve their program order with other loads/stores. Loads and Stores can reorder relative to each other. In order to preserve flow dependencies this ordering scheme requires additional hardware for dynamic memory disambiguation [1]. The hardware is simplified by only needing to compare a load address against all stores issued since the last load.

   (c) Weak Ordering (WKO): Loads and stores can reorder with any loads and stores in the program, but loads and stores to the same location occur in program order. This requires the most complex hardware. The logic must be capable of comparing all loads against all issued but not yet completed stores.

3. **Data dependency analysis**. There has been much research into the importance of precise inter and intra procedural data dependency analysis for program performance [54, 66, 69]. Memory dependencies can be resolved

[1] Preserving false dependencies is contingent on the use of memory renaming.

by dynamic disambiguation hardware and/or static analysis in the compiler; Chapter 2 presents a discussion of the types of techniques that have been suggested in the literature. This study uses the same approach as in Wall's limitation study [83] and examines the lower and upper bound for any disambiguation technique. These bounds are referred to as conservative analysis (CTA) and perfect analysis (PFA), respectively. The difference between them lies in how they handle array memory references.

- In PFA, dependency arcs exist between references to the same element of an array.
- In CTA, a dependency arc exists between each pair of references to the same array.

4. **Register and Memory Renaming**. False dependencies occur because of the imperative and sequential nature of some programming languages. These languages allow the programmer to reuse the same memory location. All false dependencies can be removed by the introduction of new variables i.e. by array privatisation [34].

5. **Synchronisation points**. The barriers that define the bounds of the reordering scope could be either synchronisation or non-synchronisation barriers. Both barriers enforce the normal barrier semantics within an instruction stream. However the synchronising barrier imposes an additional constraint that the AU must wait for the DU before issuing any further introductions. This barrier introduces false loss of decoupling (LOD) events into the program. Non-synchronising barriers allow the AU to retain some slippage ahead of the DU.

7.2 Simulation technique

The experiments use the base-line configuration for the 2-stream architecture shown in Table 7.1. This table also shows the range of architectural parameters investigated in this chapter.

The key aspects of the base-line configuration are that the decoupled memory does not include a bypass mechanism; decoupling is therefore the only mechanism used for latency hiding. There is no speculative execution, but enforcing control dependencies for loop closing branches is dependent on the reordering scope used (see Section 7.1). The instruction issue width is restricted to realistic and projected future values. The results from Chapter 6 showed that one of the most

suitable configurations for maintaining maximal throughput was (4,5). The first and second value in the brackets are respectively the AU and DU instruction issue widths.

Architecture make Components	Base-line Values	Other Values Considered
Floating operation latency (cycles)	5	
Integer/Address op. latency (cycles)	1	
Function units	(4,5)	(1,1),(2,2),(2,3)
Issue width	(4,5)	(1,1),(2,2),(2,3)
Register file	∞	
Window size	∞	
Comm. bus latency	0	
Comm. bus width	∞	
Decoupled memory	∞	
Memory bandwidth	∞	
Decoupled memory access latency (cycles)	1	
Bypass mechanism	No	
Memory differential (cycles)	60	0
Memory ordering	wko	{wko,sso,sto}
Dependency analysis	pka	{pka,cta}
Renaming	yes	no
Reordering scope	rs6	{rs0,rs1,rs2,rs3,rs4,rs5}
Synchronisation points	non-sync.	sync
Code Partition	static	

Table 7.1: Configurations for the 2-stream decoupled architecture

7.3 Experiment results

This section quantifies the performance impact of the different hardware and software design issues described in section 7.1. In the following experiments the combined issue width (CIW)for the decoupled architecture was 9 (AU and DU issue width of 4 and 5, respectively). This configuration was chosen because it is believed to be representative of future superscalar issue widths. Section 7.3.1 considers different data dependency and renaming configurations. Section 7.3.2 considers different memory ordering and barrier configurations. Each section quantifies the performance of the configurations when the memory differential is 0 and 60 cycles. An MD of 0 cycles was chosen in order to isolate the effect of the different configurations from memory latency. An MD of 60 was chosen because it is comparable to the cost of a second level cache miss [2] and it assumes a weak memory system capable of capturing no locality. In practice, for a high performance architecture, the memory system will be able to reduce the average

[2]Typical Pentium pro systems have measured L2 miss latencies of 50 cycles[8]

access time by using first and second level caches. Each section also discusses the latency effectiveness and scalability of the configuration.

7.3.1 Data dependency and renaming

Figures 7.1 and 7.2 show the measured IPC levels for different data dependency analysis and renaming configurations when the MD is 0 and 60 cycles, respectively. The suffix '+rename' and '-rename' indicates where the configuration included or excluded memory renaming. The measurements were made with weak memory ordering and non-synchronising barriers.

The 'pfa+rename' configuration in Figure 7.1 shows a large increase in IPC, from RS0 to RS1, occurring as operation reordering within leaf level loops and basic blocks is enabled. After RS1 the increasing scope for reordering operations provides smaller gains in IPC. The other significant increase occurs, from RS5 to RS6, when operations are allowed to reorder across procedure call boundaries.

Figures 7.1 and 7.2 show that even in the case of 'pfa+rename' a wide reordering scope is required to achieve high IPC when the MD is large. This can be seen clearly in the change for RS1. In Figure 7.1 RS1 has an IPC 70% of the upper limit of 7.5 IPC. Compare this with Figure 7.2 where RS1 is only 37% of the upper limit. The configuration with 'pfa-rename' exhibits a similar type of behaviour. Wider reordering scope is therefore essentially for producing higher IPC for both configurations even with perfect dependency analysis.

Reordering operations within leaf loops is insufficient at large latencies. However, it can yield a considerable percentage ($\approx$ 70%) of the IPC upper limit when latencies are small. When memory differentials are large operations must reorder across nested loops in order to achieve greater than 50% of the IPC upper limit.

Figure 7.2 shows that even with perfect analysis and renaming, it is still important to be able to reorder operations across nested loops and procedure boundaries to achieve high IPC when MD=60. This result can be seen in the large difference of 2.4 IPC (40% of the upper limit of 5.9 IPC) between RS3/RS4/RS5 and RS6.

The lower bound for any data dependency analysis technique is shown by the configuration with 'cta-rename' in Figure 7.1 and 7.2. It can be seen that in this case the scope for reordering operations is not the performance bottleneck. There is therefore clearly no benefit to increasing the scope for reordering operations beyond RS1 in this case. The consequence is that for large memory differentials the latency can not be hidden and the IPC drops to 0.5 (see Figure 7.2).

Renaming for both perfect and conservative dependency analysis only provides

additional IPC for RS3 (barriers at nested loop and procedure boundaries) and above. It can be conjectured, that the reason for this is due to the majority of false dependency arcs crossing leaf level loop boundaries. For RS1 and RS2 barriers are placed at leaf level loop boundaries. Renaming provides little benefit because most of the false arcs cross these barriers. The ordering of memory operations is therefore enforced by the barrier rather than the false dependency arc.

At large reordering scope (RS6), renaming compensates for conservative analysis when MD=0 (see Figure 7.1). However with large latencies there is little difference between the 'cta+rename' and 'cta-rename' configurations. The same is not true for perfect analysis where the gap for RS6 between 'pfa+rename' and 'pfa-rename' widens when MD is 60. This indicates, that for perfect dependency analysis renaming is more important at large memory differentials. It also shows that renaming is more important to perfect than conservative dependency analysis when MD is 60 (see Figure 7.2).

Figures 7.3 and 7.4 show the latency hiding effectiveness for different CIW for RS1 and RS6, respectively. To varying degrees all configurations show a deterioration in effectiveness as the CIW is increased. It can be seen that in both Figures the gap between the conservative and perfect analysis is large, indicating the importance of high quality dependency analysis to latency hiding.

It will also be noticed that for both configurations of perfect analysis the wider scope for reordering increases the latency hiding effectiveness. This can be seen clearly by the difference in the Figures 7.3 and 7.4 where when CIW=2 the effectiveness increases by about 20%.

For the configuration with perfect analysis, renaming improves the effectiveness of the latency hiding only when the reordering scope and issue width are large. This can be seen from the difference between Figure 7.3 and 7.4.

At large reordering scope (RS6) and perfect dependency analysis, renaming is important for improving the scalability and latency hiding effectiveness of the dual stream system. This can be seen from Figure 7.4, where for a CIW of 2 the difference in effectiveness is only 1%. However when CIW is 9 the difference from renaming is 16%. This indicates that the effectiveness and scalability of the dual stream system becomes more dependent on renaming as the CIW increases. The reduction in effectiveness for the configurations 'pfa+rename' and 'pfa-rename' is 14% and 28%, respectively.

The latency hiding effectiveness for RS6 and the configuration with perfect dependency analysis and renaming exhibits behaviour favourable to the scalability of access decoupling. Firstly when CIW $\leq$ 5 the effectiveness is greater than

90%. Secondly when CIW is 9 the effectiveness, as would be expected for a large memory differential, decreases, but only by 16%. To improve the effectiveness it will be necessary to adopt other latency hiding techniques for high CIW, such as the bypass mechanism described in Chapter 5.

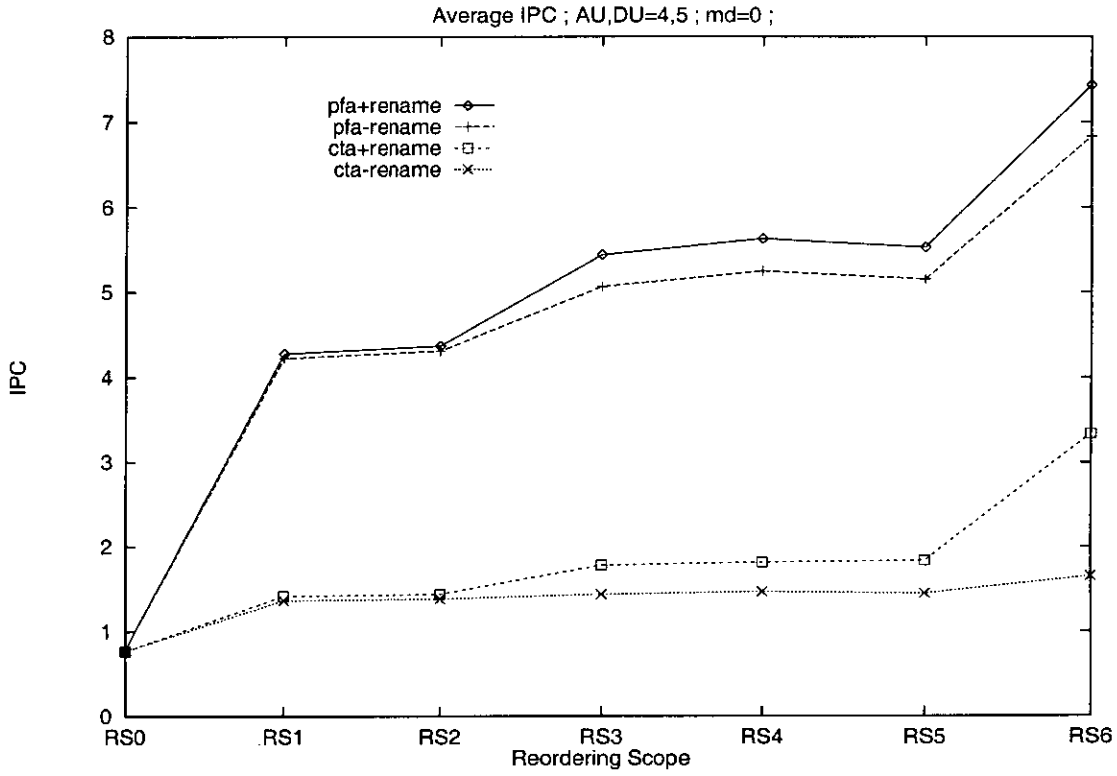


Figure 7.1: Average IPC when MD=0

7.3.2 Memory reordering and barriers

Figures 7.5 and 7.6 show the IPC for different memory ordering and barrier configurations when the MD is 0 and 60 cycles respectively. The measurements were made with perfect dependency analysis and renaming. The different barrier configurations were measured using weak memory ordering.

The most significant result is that any future dual stream system machines must support reordering of memory operations. This can be clearly seen in both Figures 7.5 and 7.6 by the large gap between the WKO and, the SSO and STO configurations. The SSO configuration has only a marginally higher IPC than the STO configuration.

There is no advantage to reordering operations beyond leaf level loops for STO and SSO configurations. Figures 7.5 and 7.6 both show that the SSO and STO configurations show no change in performance for reordering scopes beyond

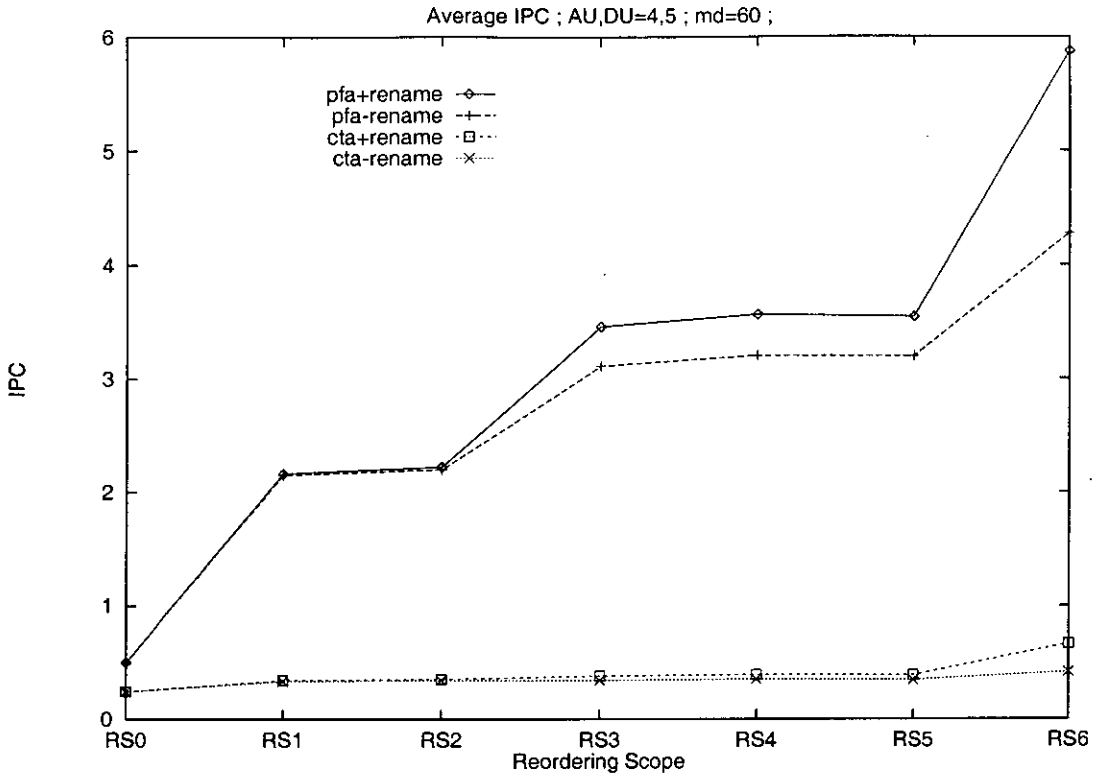


Figure 7.2: Average IPC when MD=60

RS1. Section 7.3.1 commented on the necessity of wide reordering scope in order to hide the MD of 60 cycles. It can therefore be inferred that for wide scopes, the STO and SSO configurations will have poor latency hiding capabilities because of the restrictions they impose on operation reordering. This can be seen clearly in the graph for latency hiding effectiveness in Figure 7.8.

Figure 7.6 shows for RS1 and RS2 the importance of the AU being able to continue prefetching data across leaf level loop boundaries to hide large latencies. This result follows from the large IPC difference between the configurations with synchronising and non-synchronising barriers [3].

It was also noticed that the dual stream system synchronises between nested loops and across procedure call boundaries due to data dependencies from the DU to AU. This can be inferred from the converging lines for RS3 and RS5 when the configurations have synchronising and non-synchronising barriers (see Figure 7.5 and 7.6).

Figure 7.7 and 7.8 show the latency hiding effectiveness for different CIW for RS1 and RS3, respectively. As in the previous section, it can be observed that there are varying degrees of reduction in effectiveness as the CIW increases.

[3]The line for non-synchronising barriers is coincident with the weakly ordered configuration line.

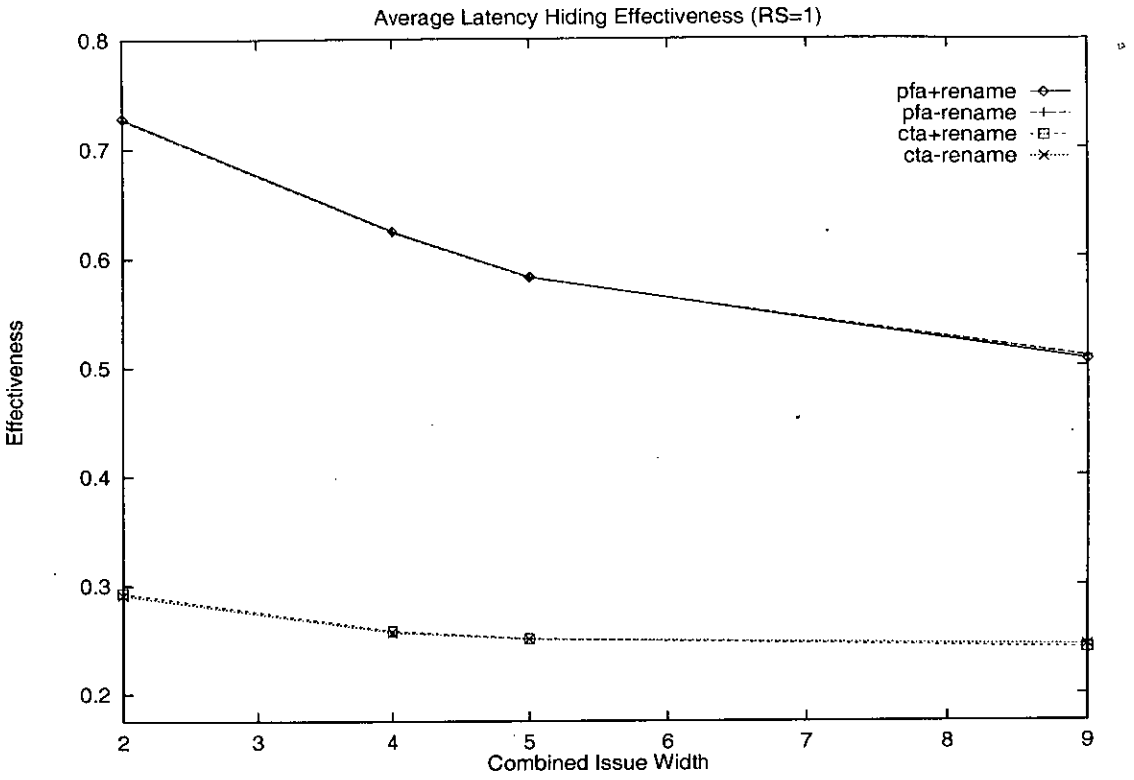


Figure 7.3: Average Latency Hiding Effectiveness (RS=1)

The major differences between Figures 7.7 and 7.8 is the increasing effectiveness of the configuration with synchronising barriers and the lack of variation in the behaviour of SSO and STO. For RS1 the effectiveness of latency hiding is poor when the configuration has synchronising barriers. As commented earlier this results from the need to allow the AU to prefetch data across leaf level loop boundaries, and from the dual stream system synchronising between nested loops boundaries. The dual stream system does not scale well with this configuration, reducing to below 40% when the CIW is 9. The effectiveness of SSO and STO is independent of instruction reordering because, the IPC does not increase with wider reordering scope and larger CIW, for these ordering schemes.

7.3.3 Individual program behaviour

The preceding analysis was based on the aggregate behaviour and therefore hides some on the individual characteristics of the programs. Table 7.2 shows IPC and LHE for different reordering schemes.

For the programs MDG, QCD2, ADM and DYFESM good inter-procedural analysis and reordering operations across nested loop is critical to achieving high IPC and latency hiding effectiveness. This can be clearly witnessed by the large

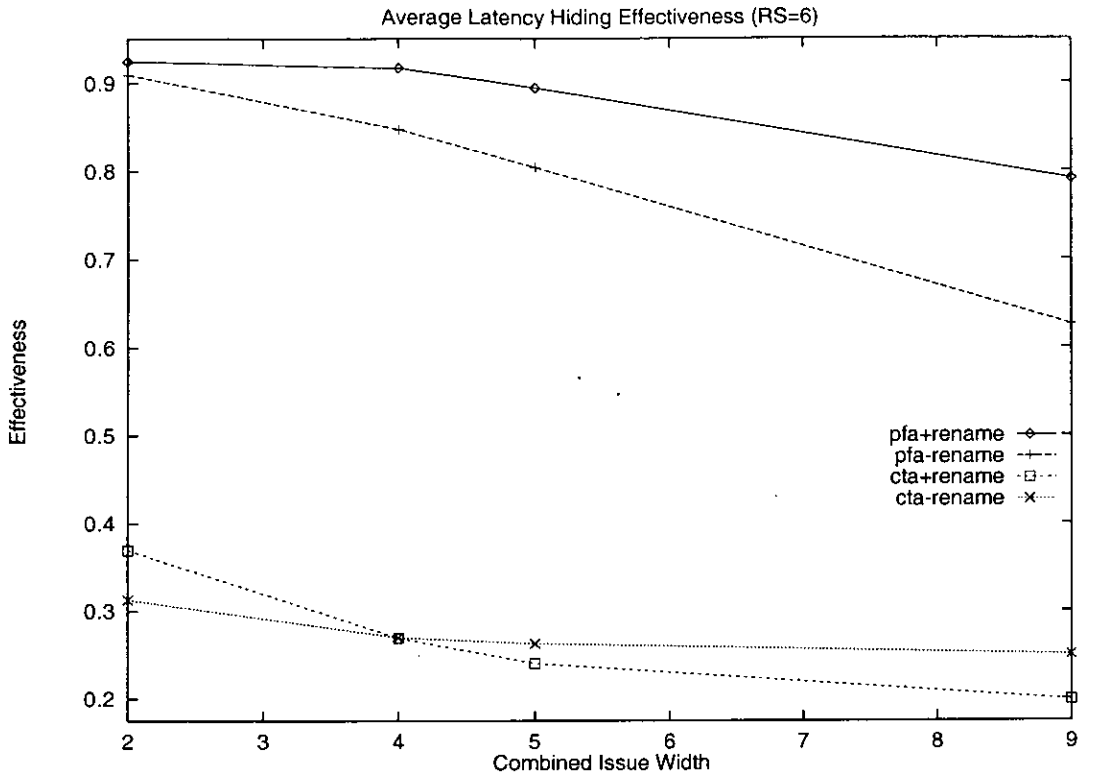


Figure 7.4: Average Latency Hiding Effectiveness (RS=6)

gap between RS3/RS4/RS5 and RS6. For example, when the memory differential is 60 cycles, ADM shows an increase of 140% in its effectiveness when reordering operations across procedure boundaries is enabled.

For the programs FLO52Q and TRFD most of the parallelism is located in the innermost loops, 87% and 74%, respectively. However to achieve a latency hiding effectiveness around 90% requires the ability to reorder operations within nested loop boundaries. In TRFD this produces a significant improvement from 50% to 99%.

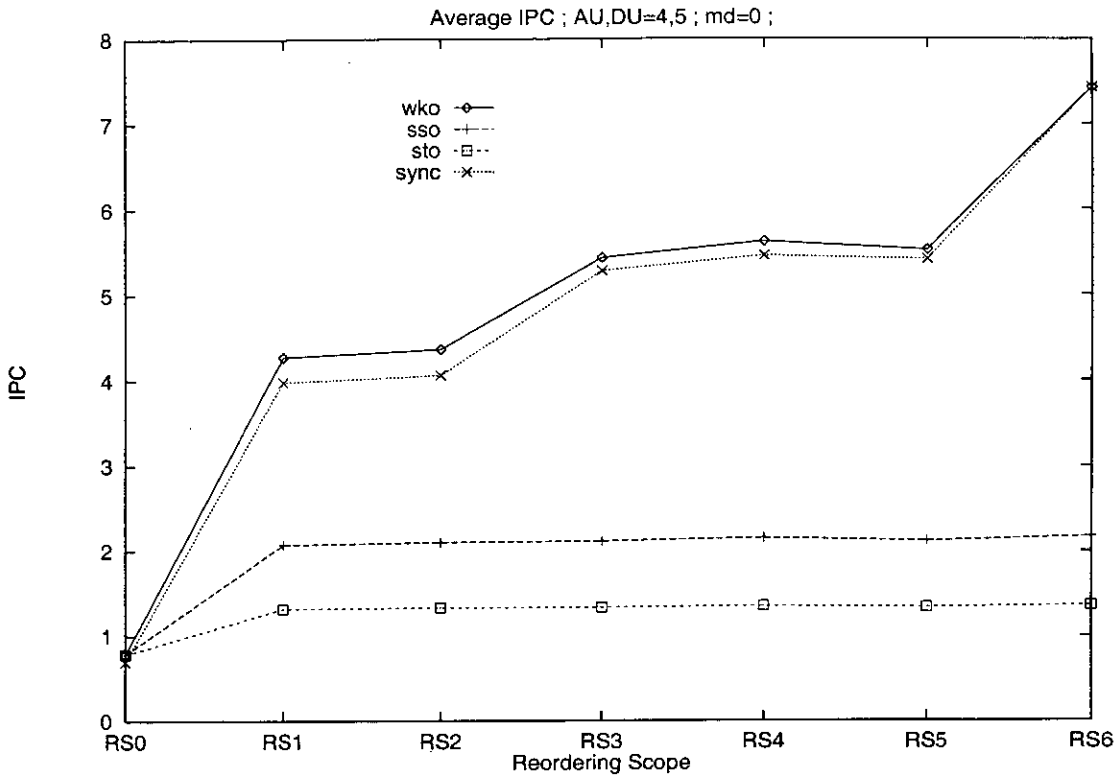


Figure 7.5: Average IPC when MD=0

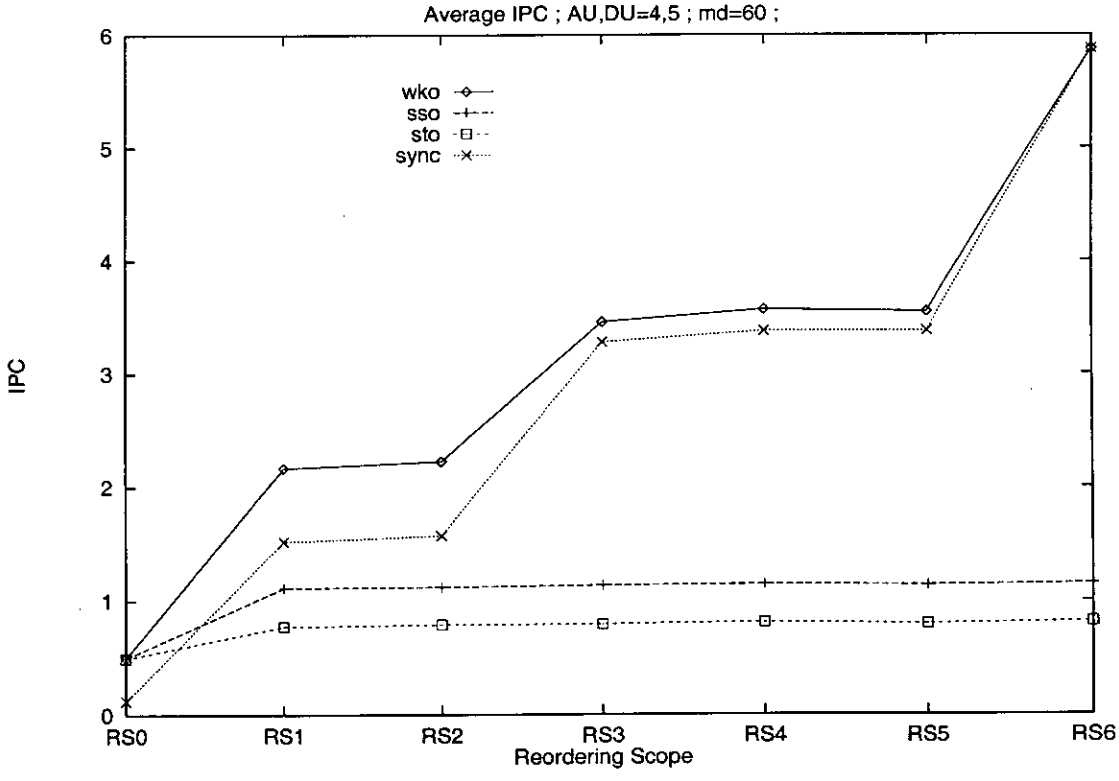


Figure 7.6: Average IPC when MD=60

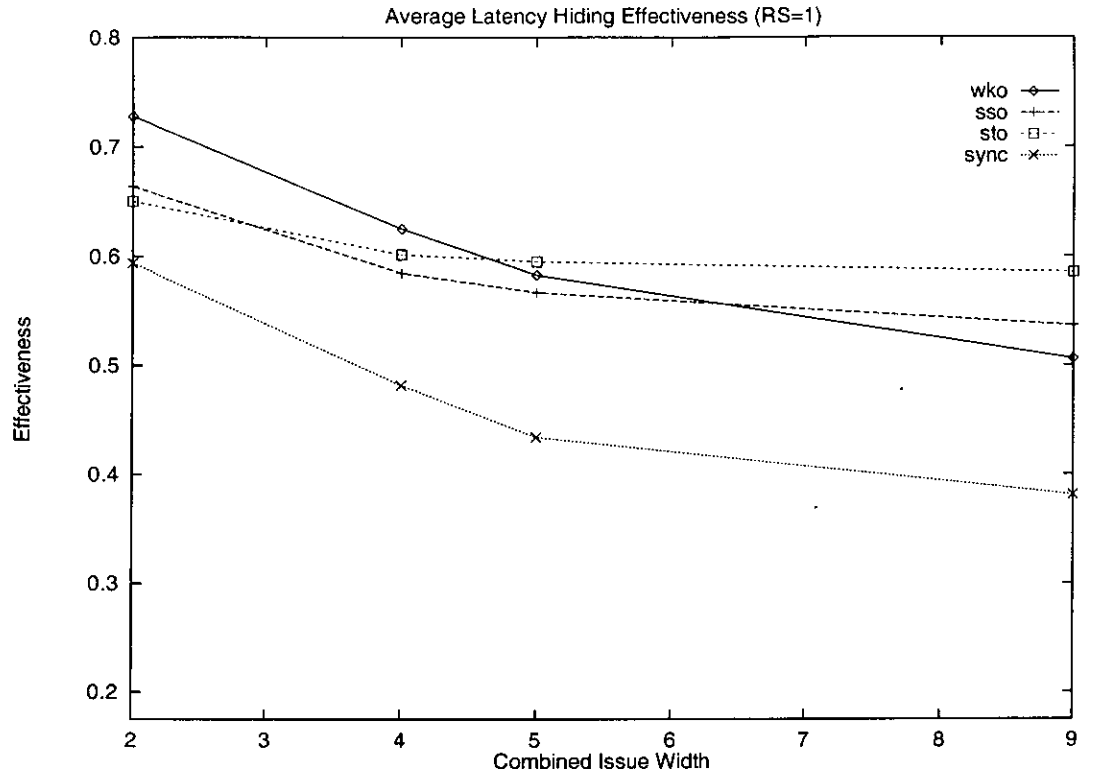


Figure 7.7: Average Latency Hiding Effectiveness (RS=1)

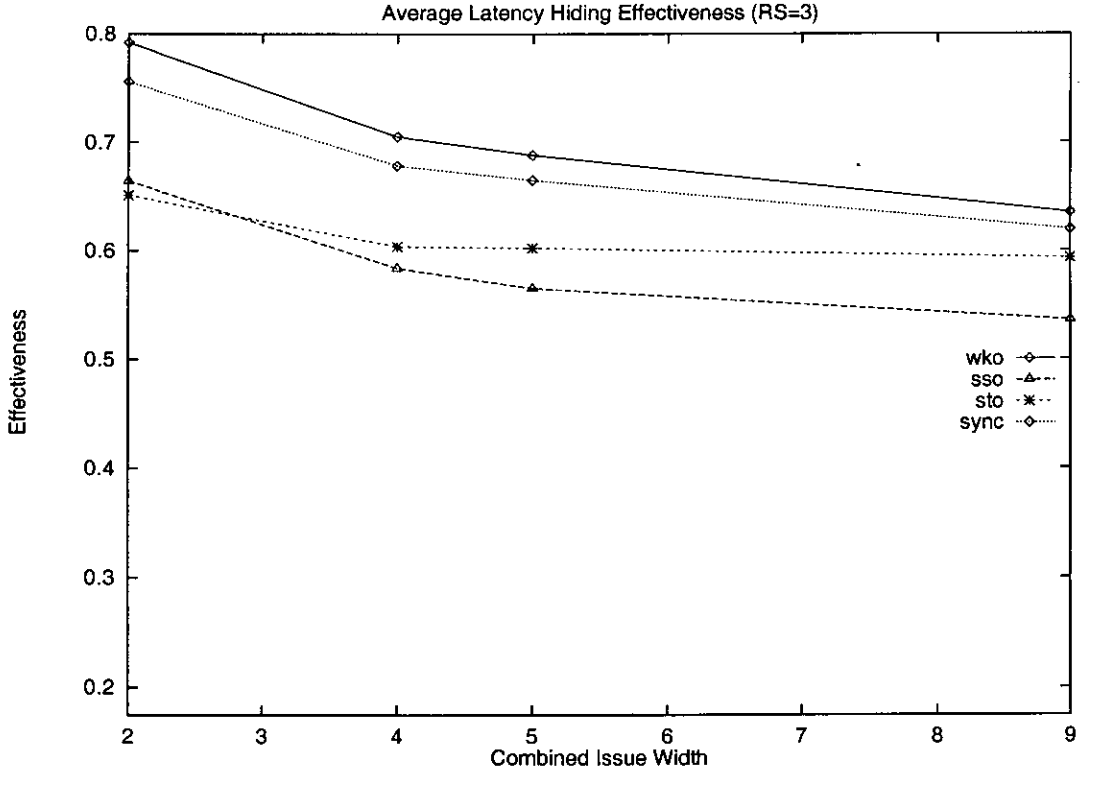


Figure 7.8: Average Latency Hiding Effectiveness (RS=3)

RS	Program	MD=0	MD=60		Program	MD=0	MD=60	
		IPC	IPC	LHE (%)		IPC	IPC	LHE (%)
0	ADM	0.8	0.5	64	DYFESM	0.7	0.6	75
1		3.9	1.6	41		3.0	1.6	52
2		4.0	1.7	42		3.0	1.6	52
3		4.4	1.8	40		5.4	3.2	52
4		4.6	2.0	42		5.4	3.2	58
5		4.6	1.8	40		5.4	3.2	58
6		8.8	8.8	100		7.6	6.6	87
0	FLO52Q	0.8	0.6	76	MDG	0.8	0.4	46
1		7.5	5.7	75		3.5	1.2	34
2		7.5	5.7	75		3.5	1.2	35
3		8.3	7.3	87		5.5	1.7	30
4		8.3	7.3	87		6.0	1.9	32
5		8.5	7.7	91		5.5	1.7	30
6		8.6	8.4	97		8.8	4.2	48
0	QCD2	0.9	0.6	64	TRACK	0.8	0.3	35
1		4.3	1.7	39		1.7	0.4	22
2		4.7	1.9	41		1.8	0.4	21
3		4.7	1.8	37		1.7	0.4	23
4		5.1	2.1	39		1.8	0.4	22
5		4.9	1.8	37		1.7	0.4	23
6		8.1	4.5	55		1.9	0.4	22
0	TRFD	0.7	0.6	86	Average	0.8	0.5	
1		6.1	3.0	50		4.3	2.2	
2		6.1	3.0	50		4.4	2.2	
3		8.2	8.2	99		5.5	3.5	
4		8.2	8.2	99		5.6	3.6	
5		8.2	8.2	99		5.5	3.5	
6		8.2	8.2	100		7.4	5.9	

Table 7.2: IPC and LHE for different reordering scope; dependency analysis is perfect with renaming

7.4 Design implications for the dual stream system

In Section 7.3.2 it was shown that the reordering of memory operations is critical for hiding large memory latencies and achieving high IPC. The implication of this result is that the decoupled memory (described in Section 3.1) must, like the ARB [40] be capable of supporting multiple accesses, speculative loads and stores, and dynamic memory disambiguation. The ARB could be placed in the decoupled memory to support the aggressive reordering of memory operations of a dual stream architecture [4]. However, by itself, the ARB is not sufficient to support an out-of-order dual stream architecture. This is because the order in which operations are executed on the AU, and the order in which data arrives in the decoupled memory, must be communicated to the DU. There also has to be additional support for waking up operations on the DU when data arrives in the decoupled memory.

One of the postulates of this thesis is that the decoupling paradigm can simplify instruction issue logic. Therefore, the central design requirement for this additional wakeup logic is that it should not introduce greater complexity into the instruction issue logic. This section proposes a mechanism that uses the sequence numbering of memory accesses to support out-of-order decoupled execution.

Although operations are executed out-of-order, they are fetched and decoded in-order. In a similar way to the ARB, a sequence number is assigned to each AU and DU operation that accesses the decoupled memory. This results in the AU and DU operations involved in a decoupled load having the same sequence number. This sequence number is referred to as the order number, and is used by the DU to detect if the data is available in the decoupled memory. The order number has a maximum limit that restricts the number of accesses that can be in-flight at any one time. The order number is a logical sequence modulo the maximum limit.

7.4.1 Out-of-order decoupled loads

The diagram in Figure 7.9 shows how the order number can be used to support the out-of-order execution of a decoupled load. The diagram shows a load data buffer that is indexed by the order number. It also shows, in each of the decoupled units, an order manager queue to control the assigning and releasing of order numbers.

[4] Interestingly, the ARB includes a mechanism to forward the values of store operations to waiting loads; this is similar to the bypass mechanism in the optimised decoupled memory. Section 9.1.6 has shown the benefits of this mechanism for a decoupled architecture.

The number of entries in the order manager queue and load data buffer are equal, so that no two in-flight accesses can map to the same slot in the buffer.

To illustrate how the load data buffer and order manager queues support out-of-order accesses in a decoupled architecture, Diagram 7.9 shows the different stages of a fully decoupled load [5] from an address X. At each of the labelled stages in the diagram, the following operations would occur:

- In Stage 1 the address operation is decoded and placed in the AU instruction window. During this stage, the operation is assigned the order number $n+1$, and the tail is incremented (modulo the size of load data buffer) to point to this latest instruction.

- In Stage 2 the address is sent to the decoupled memory with its order number. Note, that because the load data buffer has the same number of entries as the order manager queue, a slot is implicitly reserved for the retrieved data in the load data buffer.

- In Stage 3, the accessed data is placed in the decoupled memory where it waits to be fetched by the DU.

- In Stage 4, the DU decodes the fetch of X and assigns it the order number $n+1$. The order number is then tagged with the instruction window slot ID where the fetch operation resides, and is sent to the load data buffer. The result of the successful lookup is communicated, using the slot reference (thereby avoiding an associative lookup), back to the instruction window and the fetch operation is flagged as ready to execute.

- In Stage 5, the fetch operation executes and the data is retrieved successfully from load data buffer.

- In Stage 6, once the data has been retrieved from the load data buffer the order manager queues on both units are notified that this order number can be released (marked as 'R'). The header is advanced when the order number it points to is marked for release.

If the load is not fully decoupled (the DU instruction is decoded and placed in the instruction window prior to the data arriving in the decoupled memory) then the following operations occur. The load data buffer detects that the data hasn't arrived and places the slot reference in the data area. When the data arrives from

[5] One in which the data is already resident in the decoupled memory when the DU operation arrives in the instruction window

main memory, the load data buffer uses the reference to inform the DU that the fetch operation can be now flagged as ready to execute. In parallel, the retrieved data is also written to the data area.

The advantage of using the slot reference is that it does not require expensive wakeup logic to broadcast a result to every operation in the instruction window [6].

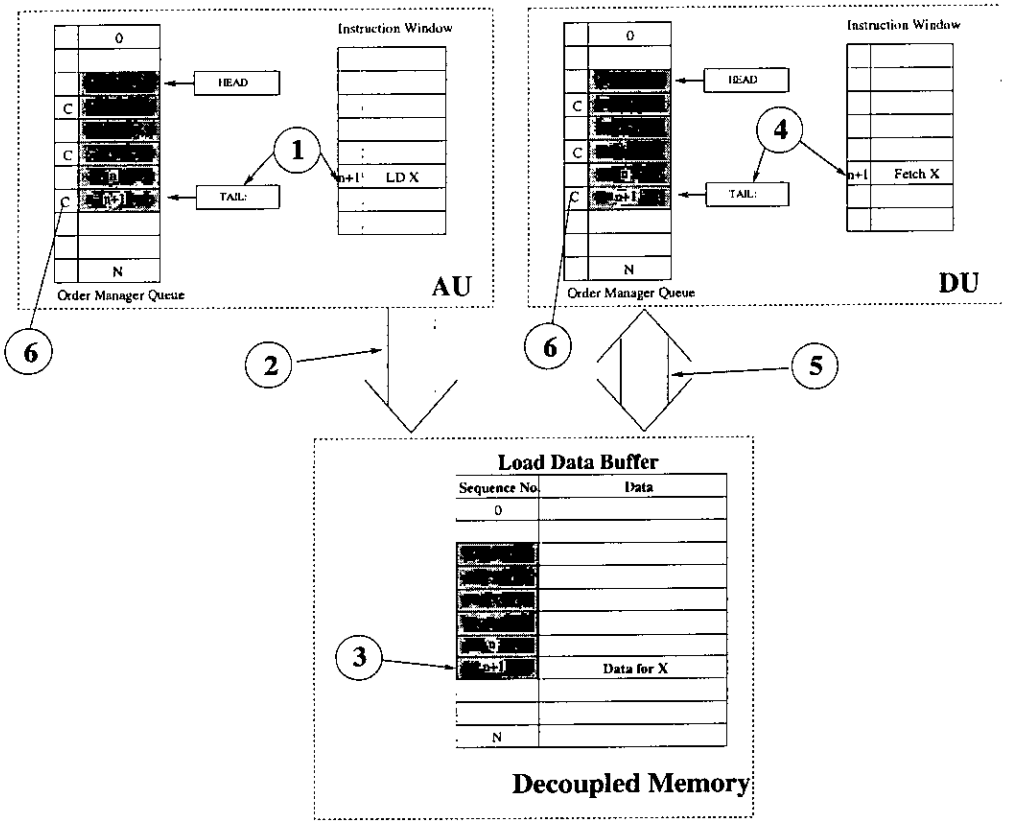


Figure 7.9: Support for out of order memory access in a dual stream system

7.4.2 Out-of-order decoupled stores

An out-of-order decoupled store can also be implemented using the ARB. The only additional complexity is the hardware support to match DU data with the appropriate AU address. This will require an associative lookup on the order sequence number of waiting store addresses in the decoupled memory (not shown in Figure 7.9).

[6] Palacharala [64] showed that the associative wakeup logic required in the issue window introduced delays that increased quadratically with the window size.

7.5 Summary

This chapter has identified those design issues that determine the degree of reordering and investigated their relationship to the latency hiding effectiveness, scalability and IPC of the dual stream system; where scalability is defined as the variation of the latency hiding effectiveness with the CIW.

The results from this chapter have shown the importance of memory ordering. The SSO and STO schemes have been shown to restrict the parallelism in the dual stream system. For this reason this chapter has proposed a mechanism to support out-of-order accesses in decoupled architecture. The key component in the mechanism is a load data buffer that serves two functions: to hold data retrieved from the main memory; and to hold references to slots in the DU instruction issue window. The latter functionality is required for DU fetches that can not be serviced because the data is not resident in the decoupled memory. The advantage of the proposed technique is that it can be combined with an existing mechanism (the ARB [40]) while only introducing minimal complexity into the instruction issue logic.

Interestingly, the SSO scheme was used in a previous decoupled architecture [24], but operations were only allowed to be reordered from within leaf level loops. This chapter has shown, that for the dual stream system this is the optimal configuration of reordering scope with the SSO scheme. Results show (see Section 7.3.2) that providing a wider scope for reordering operations beyond leaf level boundaries yields minimal benefit for the SSO scheme.

Renaming is only important to the latency hiding effectiveness of the dual stream system for wide reordering scope and large CIW (> 5).

For a configuration with perfect dependency analysis, weak memory ordering and renaming, behaviour favourable to the scalability of the dual stream system has been observed. The latency hiding effectiveness has been shown to be greater than 90% for CIW ≤ 5 and only decrease by 14% when CIW is large (9-way issue).

This chapter has also tried to quantify the effect of differing complexity of parallel optimisations; where complexity has been modelled in terms of wider reordering scope. The results show that when memory differentials are small reordering operations in leaf level loops is sufficient to achieve 70% of the IPC upper limit. For large memory differential, operations must be able to reorder within nested loop boundaries to achieve greater than 50% of the IPC upper limit.

Wide reordering scopes can offer sufficient operations to hide latency but are dependent on the memory ordering scheme and the data dependency analysis.

For configurations with STO, SSO and conservative analysis the benefits of wide reordering scope can not be realised, due to the memory ordering scheme and dependency analysis being the bottleneck.

In three of the programs, reordering operations across procedural boundaries is critical to extracting ILP. For the same programs it has also been shown to be necessary for tolerating large memory latencies. Highly parallel programs have been found to have high ($> 87\%$) latency hiding effectiveness when operations are only able to reorder within nested loops.

For configuration with narrow reordering scopes, where barriers exist at leaf level loops, results have shown that it is important to allow the AU to continue prefetching data after the barrier. Synchronising barriers are observed to reduce performance especially when the MD is large.

Chapter 8

Reducing Hardware Complexity through Decoupling

It can be speculated that future high performance microprocessors will improve performance by extracting higher degrees of instruction level parallelism. In superscalar architectures parallelism is exploited by reordering instructions within an instruction window and issuing multiple independent instructions per cycle. However as processor speeds increase and issue widths get larger the cost of a main memory access is becoming relatively more expensive. One solution is to hide memory latency by *data prefetching*.

Data prefetching is a technique that hides memory latency by overlapping access and data operations. Data prefetching can be implemented in either hardware [41] and software [16] or a hybrid [18] of both schemes. However as memory accesses become relatively more expensive the number of independent overlapped instructions required to hide access times increases. Larger instruction windows are therefore required to detect independent instructions that can execute in parallel with memory access operations.

The pressure to increase window sizes is also driven by the goal of providing ever larger issue widths. However, large windows and issue widths introduce greater complexity in window issue logic. Palacharla [64] has shown that delays in the issue logic vary quadratically with window and issue width size. Since delays in issue logic will be critical to processor clock frequency there is a need to consider architectures that simplify issue window logic.

To solve the window complexity problem some architectures use separate microclusters. Microclusters may share, or have a dedicated instruction window, but each has its own register file and function units. This design simplifies window logic by flagging instructions for execution on particular microclusters. This reduces the size of the instruction window but can limit the number of instruc-

tions issued per cycle. Other designs, like the MISC [81], Multiscalar [75] and PEW [51] architectures, use decentralised control logic to simplify the window hardware, and increase ILP without effecting clock speeds.

Since in principle the same level of prefetching in an decoupled machine could be achieved with a single stream, out-of-order superscalar architecture; the question is "why should designers consider using the decoupling paradigm?"

Memory latencies are typically 20-50 cycles whereas arithmetic function latencies are 2-5 cycles (excluding divide and intrinsics). A system could easily tolerate a small degree of out-of-order execution amongst arithmetic operations provided loads could *slip* by a large amount with respect to arithmetic operations. This slippage between arithmetic and load operations is exactly what occurs in a decoupled machine. In other words, there can be small distinct instruction windows for arithmetic and access operations, provided the latter can slip by a large amount with respect to the former. To illustrate this idea section 8.1 introduces the concept of the *effective single window*. The effective single window is the minimum size of window required by a processor with a single window to have the same instructions in flight as the decoupled machine. In answer to the question posed in the paragraph above, it is believed that a decoupled architecture's decentralised logic would produce high latency tolerance while reducing the complexity of reordering hardware.

The disadvantage of decoupled architectures is the extra communication overhead, and the smaller issue width and instruction window available to each of the instructions streams. However, communication hardware can be implemented cheaply through queues, and is not critical to processor clock speeds. Provided the frequency of communication is low and the bandwidth is high, then the impact of the communication overhead should be minimal. The OCTAVE compiler reduces communication by duplicating operations on both units [1]. Current advances in fabrication technology mean that with decreasing features sizes a single chip multiprocessor could be built with an inter-unit communication latency of a few cycles [2]. However, narrower issue widths and smaller window sizes may increase contention and limit reordering; this will reduce the latency hiding effectiveness and possibly annul any of the benefits derived from simpler issue hardware.

This chapter justifies renewed interest in decoupling by comparing a single and dual stream out-of-order superscalar architecture. Section 8.3 considers the effect of issue width on both machines. In order to isolate its effect on performance all

[1] Though this optimisation can cause code expansion.

[2] Chapter 9 provides a series of simulation studies to analyse the effect of communication cost on a decoupled machine.

resources are relaxed and unlimited reordering is permitted. Using an analytical model and simulation studies, this section compares the performance of a single and dual stream systems. The results show that although single stream systems achieve *slightly* higher IPC, on average it requires *much* larger issue widths and window sizes. This suggest that any performance loss could be more than compensated for by simplifying the control hardware. Section 8.3 also shows that the code expansion introduced by OCTAVE's static partitioning has little effect on these findings when issue widths are large.

Section 8.4 compares the relationship between window size and memory latency for the decoupled and a single stream superscalar machine. It also evaluates the size of window required by the single stream architecture to achieve the same performance as the dual stream architecture.

In the remaining sections; Section 8.1 discusses the notion of the *effective single window size* to help explain some of the chapters findings and Section 8.2 describes the simulation technique. Section 8.6 concludes by summarising the chapter findings.

8.1 The effective single window

An advantage of the decoupled machine is that the dynamic slippage between the window of instructions on the AU and DU means that the effective single window size can be greater than the sum of the individual units' window sizes. Figure 8.1 illustrates the idea of the effective single window. The diagram shows the streams for the AU, DU and a single instruction stream. For the single instruction stream, the instructions are shown in program order (with later instructions appearing further down the page), labelled with the units on which they execute in the decoupled machine. The diagram shows that, due to the dynamic slippage between the units, the AU is executing instruction further into the instruction stream than the DU. The effective single window is the minimum size of window required to buffer all instructions from the oldest DU instruction to the youngest AU instruction.

8.2 Simulation technique

In the dual stream system, loads and stores are executed as one instruction on each of the units. On the single stream architecture, loads and stores generate a prefetch and an access operation. Integer and address computations have a 1 cycle cost. Floating point operations take 5 cycles to complete.

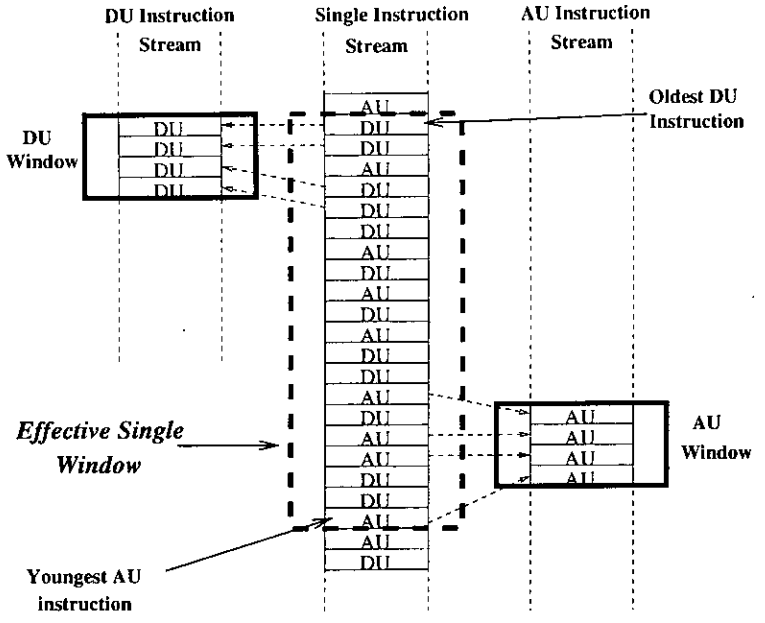


Figure 8.1: Effective single window

There is no speculative execution but it is assumed that loop closing branches have been removed by optimisations like loop unrolling and branch prediction. Data dependency analysis is perfect and false dependencies are removed by renaming. The purpose of examining such an ideal case is to provide the best opportunity for prefetching data, to have high instruction level parallelism (ILP) and to place the greatest pressure on the latency hiding mechanism.

The results from Chapter 6 are used to select instruction issue widths. The values selected were (4,5) for the decoupled and an issue width of 9 for the single stream architecture.

8.3 Issue width

It can be argued that the degree of prefetching of an access decoupled machine could be achieved with a single stream architecture. If this is the case why introduce the additional complexity required to support asynchronous instruction streams. This thesis postulates that the decoupled architecture will be able to provide lower but comparable performance but with smaller window size and issue widths; this will simplify the instruction issue logic and reduce the clock period.

The disadvantage of the decoupled architecture is that it uses a smaller issue width for each stream, potentially increasing slot contention and reducing the peak issue rate for data and access operations. One would anticipate that the net

result would be a drop in the IPC for a decoupled architecture. A key question is whether the trade-off between simpler hardware is offset by the reduction in IPC. To answer this question, the approach adopted in this chapter is to quantify the reduction in IPC for the decoupled architecture. The rational behind this approach is that we can speculate a small reduction in IPC will be compensated for by a reduction in cycle times. However, a large reduction in IPC will require analysing the hardware complexity of the two architectures in greater detail.

In the first study, the architectures were given the same combined issue width but allowed unlimited instruction window size. In order to isolate this comparison from other factors, the same number of operations were executed on both the dual and single stream systems. Although there is some duplication of operations in the case of the dual stream system, the increase is on average 9.2% (see Table A.1). It is believed that the effect of this code expansion will decrease at the large issue widths being considered in this chapter [3]. The programs were executed with perfect data dependency analysis, renaming, weak ordering and non-synchronising barriers.

Tables 8.1 and 8.2 show the average IPC measured for the 7 programs when the reordering scope was RS3 and RS6 respectively [4]. The tables also show the reduction in IPC for the dual stream system and the latency hiding effectiveness of the two systems. The column labelled CIW is used to denote the *combined issue width* for dual stream and the issue width for single stream. In the case of the dual stream the combined issue width is the sum of the AU and DU issue width. The experiments only looked at the cases when the dual stream combined issue width was equal to single stream issue width.

It can be seen that in all cases the dual stream system's IPC is lower than that of the single stream system. The reduction in performance is due to inflexible scheduling during *synchronising* and *startup* phases. When the decoupled machine is in the state where the AU is waiting for a result from the DU, its issue slots are being wasted. However, for the single stream system the full issue width is available at all times. During the startup phase after a synchronisation point, the single stream system is able to initiate more accesses whilst the DU waits for the first value to be returned from the memory system. It would be expected therefore that as the number of synchronising points decreases the reduction in IPC would be smaller. This can be seen in the difference between tables 8.1 and 8.2, where for a combined issue width of 9 and memory differential of 0,

[3] Code expansion and other design issues unique to decoupling are considered in Chapter 9

[4] In RS3, intra nested loop reordering is permitted, but operations can not migrate across procedure call and nested loop boundaries. In RS6, there are no restrictions on reordering.

the reduction in IPC has dropped from 9% to 6%. This is due to RS3 reducing the slippage between the units and hence increasing the number of synchronising points in the program.

The difference in IPC also varies with larger combined issue width. This is due to the fact that when the combined issue width is 2 the AU and DU have an issue width of 1, effectively serialising the code for each of the instruction streams (some reordering will be possible but it will be small). When the combined issue width is increased to 9, the benefit accrued to the single stream system by the extra issue slots, available during synchronisation and startup phases, is relatively smaller.

The tables also show that the latency hiding effectiveness is always marginally smaller for the single stream architecture. It is believed that this is due to the memory accesses (operations in the dual stream system would be executed on the AU) having to contend with data computations (operations in the dual stream system that would have been executed on the DU) for issue slots.

A large memory differential also reduces the performance difference between dual and single stream architectures. Large memory differentials reduce the parallelism in the program and limit the advantage gained by the single stream system during synchronising and startup phases on the dual stream system.

This analysis permitted unlimited lookahead through the unconstrained instruction window. An estimate of the average window size can be computed using an analytical model, the measured average perceived load latency and the operation breakdown shown in Table A.2. The average length of time for an operation to execute is given by

$$\overline{t_{op}} = \overline{pll} * \alpha_{load} + 1 * \alpha_{store} + 5 * \alpha_{float} + 1 * \alpha_{integer} \tag{8.1}$$

where the terms α_{load}, α_{store}, α_{float} and $\alpha_{integer}$ denote the fraction of load, store, float and integer operations, respectively. The average length of time any operation will wait in the instruction issue window for a result will be equal to $\overline{t_{op}}$. The number of instructions decoded per cycle is less than or equal to the the issue width (IW); the inequality applies because branches will cause dependent instructions to wait for the target address to be calculated, delaying their execution by the fetch/decode mechanisms. The average window size is therefore given by

$$\overline{w} \leq \overline{t_{op}} * IW \tag{8.2}$$

In the following experiments the fetch and decode is assumed to be perfect so that the inequality becomes an equality in this equation. Table 8.3 shows the computed average window size for the dual and single stream architectures when RS=6. This table shows that the single stream system requires a larger instruction window than the decoupled architecture. The average ratio of the single to dual stream window size is 1.7 and 2 when the combined issue width is 5 and 9, respectively. This also shows that the ratio increases with larger combined issue width.

Table 8.2 shows when RS=6, that for a combined issue width of 5 and 9 the average degradation in IPC is 7% and 5%, respectively. Palacharla [64] has shown that delays due to window logic increase quadratically with instruction window size and issue width. He found that the performance degradation between a 2x4-way clustered and a single 8-way architecture was only 2-12%. From his hardware complexity analysis he was able to conclude that the 2x4-way architecture would have a 25% faster clock speed, resulting in an average performance improvement of 16%. It is therefore anticipated that the 7% to 5% reduction in IPC would be more than compensated for by the simpler issue logic in the dual stream system.

By executing the same number of operations in the single and dual stream architecture the effect of code expansion was isolated from the experimental findings. Table 8.4 shows the average speedup of the single over the dual stream architecture for the cases when the former was without and with the duplicated operations of the dual stream architecture. Table 8.4 shows that with small combined issue widths of 2 instructions the single stream, without code expansion, has a significantly higher IPC than the dual stream architecture. To achieve equivalent performance would require the dual stream to have a clock frequency 17-20% higher than the single stream architecture. However, with an issue width of 9 and an MD of 60 cycles, the speedup of the single stream is reduced to between 4% and 6%. It is expected that such a small loss in performance would be compensated for by the reduction in clock speed.

Table 8.4 also shows that with a CIW of 2, the speedup with and without code expansion is 5.2% and 3.5% higher for an MD of 0 and 60 cycles, respectively. For a CIW of 9 the speedup is only 2.8% and 2% higher, for an MD of 0 and 60 cycles, respectively. It can be concluded therefore, that the effect of code expansion is only small for large CIW. The assumption made above that code expansion is not significant only applies for large combined issue widths. This result has been inferred from simulations with unlimited window size. A discussion of the effect of code expansion for a restricted window size is postponed until Section 9.1.2.

Instruction Stream	CIW	Average IPC		Latency Hiding Effectiveness (%)
		md=0	md=60	
Single	2	1.86	1.44	77
Dual	2	1.56	1.24	79
IPC reduction (%)		0.3 (16)	0.2 (14)	
Single	5	3.93	2.59	66
Dual	5	3.56	2.45	69
IPC reduction (%)		0.37 (9)	0.14 (5)	
Single	9	5.97	3.7	62
Dual	9	5.44	3.46	64
IPC reduction (%)		0.53 (9)	0.24 (6)	

Table 8.1: Comparison of single and dual instruction streams when RS=3

Instruction Stream	CIW	Average IPC		Latency Hiding Effectiveness (%)
		md=0	md=60	
Single	2	1.92	1.76	92
Dual	2	1.68	1.55	92
IPC reduction (%)		0.24 (13)	0.21 (11)	
Single	5	4.54	4.00	88
Dual	5	4.18	3.74	89
IPC reduction (%)		0.36 (8)	0.26 (7)	
Single	9	7.94	6.16	78
Dual	9	7.43	5.87	79
IPC reduction (%)		0.51 (6)	0.29 (5)	

Table 8.2: Comparison of single and dual instruction streams when RS=6

8.4 Window size and memory latency

The conclusions drawn from the previous section were that a decoupled system could achieve slightly lower, but comparable performance to a single stream architecture, and that this could be achieved with a window of half the size. However, these results were based on an analytical model and simulation studies of architectures with an unlimited reordering window. This section describes the effect of window size and memory latency on the dual and single stream architectures. The experiments simulated all the programs from Table 5.5, but for the purposes of this chapter three representative programs are selected that exhibit the range of observed behaviour [5]. The three selected programs were FLO52Q, MDG and TRACK. Figure 8.2 shows the latency hiding effectiveness of all seven programs

[5]Results from the other programs are shown in Appendix E

Program	CIW=5		CIW=9	
	decoupled au,du	single	decoupled au,du	single
ADM	4,30	41	6,35	75
DYFESM	2,20	65	5,65	111
FLO52Q	4,31	54	5,24	95
MDG	7,27	41	14,43	75
QCD2	4,33	36	8,52	66
TRACK	15,45	56	30,76	100
TRFD	2,24	41	4,47	78

Table 8.3: Average window size when RS=6

CIW	md=0		md=60	
	with	without	with	without
2	1.14	1.20	1.13	1.17
5	1.08	1.13	1.07	1.09
9	1.07	1.10	1.04	1.06

Table 8.4: The effect of code expansion on speedup when RS=6

when the window size is unlimited and the memory differential is 60 cycles. It can be seen there are three bands in which the programs are highly (80-100%), moderately (40-60%) and poorly (< 40%) effective at hiding latency. It can be seen that the three programs fall within each of the bands.

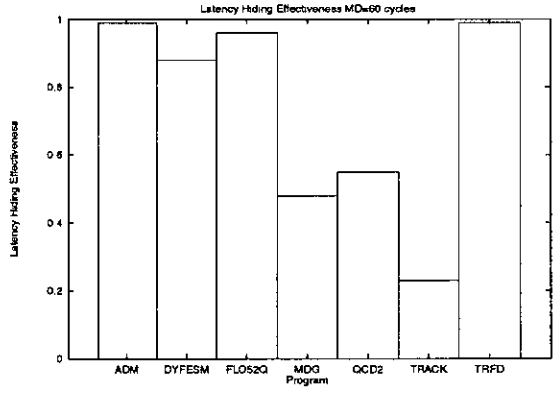


Figure 8.2: Latency hiding effectiveness MD=60 cycles

Figures 8.3, 8.4 and 8.5 show the variation in speedup with window size for the dual and single stream architectures when the memory differential is 0 and 60 cycles. When MD is 0 it can be seen that for small window sizes the dual stream system performs better than the single stream system with the same window

size. This is due to the dual stream system having two windows for reordering operations compared to one for the single stream. This means there are fewer resource conflicts for window slots and greater scope for reordering operations. It can also be noticed that the graphs show diminishing returns for increasing window size; once window sizes are above 10 instructions, doubling the size does not double the speedup. All the programs reach a cut-off point for window sizes between 40 and 80 instructions when the single stream system performs more effectively. This is due the benefit of the larger instruction issue width available to the single stream. This benefit is only realised once the instruction window is large enough to utilise the available issue width.

Figures 8.3, 8.4 and 8.5 show that once MD reaches 60 cycles there is no cut-off point at which the single stream system performs better than the decoupled machine. This results applies even for very large windows of 100 instruction slots. The difference between the performance of the two machines must be solely due to the more effective data prefetching of the decoupled machine. Operations on the single stream system, which on the decoupled system would have been executed on the DU, are causing address computations to execute later, reducing the pipelining of memory accesses and decreasing the effectiveness of data prefetching. The difference in performance between the two machines also depends on the type of program. For FLO52Q which is highly parallel, the gap between the decoupled and single stream system is large. However, for TRACK which has little parallelism, there is little difference between the two architectures.

It can be stated therefore that for all the simulated programs the decoupled machine is more effective at hiding large memory latencies than the single stream architecture. The difference in performance is dependent on the available parallelism and decoupling in the program. Programs that decouple well show the largest improvement in performance for the decoupled machine.

Figures 8.6, 8.7 and 8.8 show, for a range of memory differentials, the *equivalent window ratio*. This term is the ratio of the single stream and decoupled window sizes that yield equivalent performance. The ratio was derived by projecting from the decoupled system to the single stream graph in Figures 8.3,8.4 and 8.5. The graphs show the way in which the ratio varies as a function of the memory latency. It can be seen that as latencies approach 60 cycles the ratio gets larger. This is due to the more effective data prefetching of the access decoupled machine. As the memory latency increases, the DU waits longer for data to arrive and the slippage between the two units grows. This means that the effective single window size (see Figure 8.1) for the decoupled machine gets

larger. In order for the single stream system to achieve equivalent performance, it requires a correspondingly larger window.

The graphs in Figures 8.6, 8.7 and 8.8 also show that as the decoupled window size is increased the equivalent window ratio reduces. This is due to the single stream architecture being able to reorder operations to a similar degree as the decoupled machine, and also the benefits of the larger issue width.

Significantly, it can be observed that for a realistic decoupled window size of 30 instructions and a memory latency of 60 cycles, the equivalent window ratio is dependent on the program, but is in the range 2.5 to 5. Experiments with the other benchmark programs shown in Table 5.5 have also been found to fall within this range (see Appendix E). Larger windows introduce extra hardware complexity and longer window logic delays. It can be stated therefore that the decoupled machine requires significantly smaller instruction windows and hence simpler window logic.

For smaller memory latencies of 20 cycles the range for the equivalent window ratio is smaller, between 2.2 and 3.5 for 7 of the programs. However, for TRACK the ratio drops below 2 to 1.9. This is due its low parallelism which even with unlimited window size does not increase above 2 IPC. The effective single window is clearly unable to provide any additional advantage because of this low parallelism.

Having shown that the dual stream system performs consistently better than the single stream architecture, a comparison is made between the latency hiding effectiveness of the decoupled machine against a perfect latency hiding technique (one in which all the memory differential is hidden). Table 8.5 shows the measured LHE for different window sizes when the memory differential is 60 cycles.

The results show that when window sizes are small, increasing the window size causes a reduction in the LHE. This is due to the extra parallelism on the DU placing greater pressure on the memory system. The AU window is not yet large enough allow the AU to pipeline accesses sufficiently to hide the latency. However there eventually comes a point when the larger window size allows more operations to execute in parallel and the LHE starts to improve. For MDG and FLO52Q that point occurs at 40 and 50 instructions respectively. This result suggests that for realistic window sizes (1 to 30 instructions), increasing the window size will result in the latency hiding mechanism of the dual stream system performing less effectively. Table 8.5 also shows that even large window sizes do not approach the latency hiding effectiveness of a decoupled machine with unlimited resources.

These findings show that for realistic window sizes the dual stream system

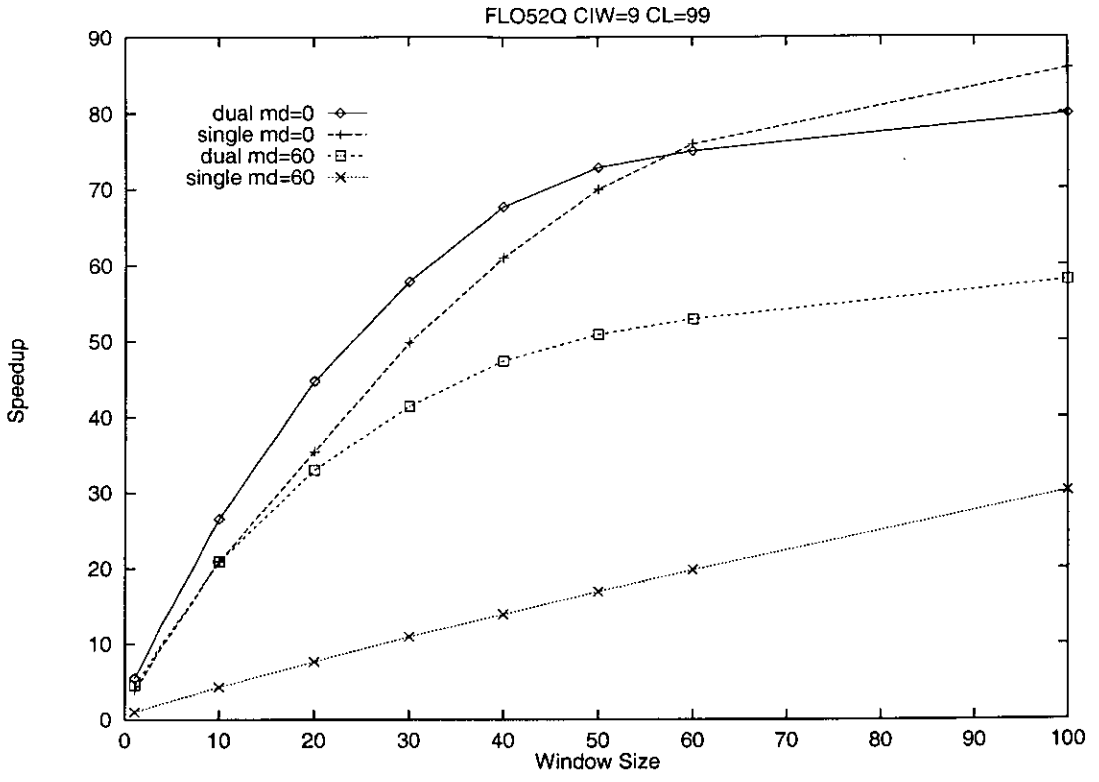


Figure 8.3: Speedup of FLO52Q for varying window size

can hide latencies better than single stream architecture but that as the window size increases its effectiveness at hiding latency deteriorates. This illustrates the tensions that exist between having greater parallelism and the access decoupling mechanism. As the window size gets larger, the instruction level parallelism increases and the execution times fall. However, the extra parallelism places greater pressure on the decoupling mechanism resulting in a decrease in LHE. The result is the memory latency contributes more to the critical path time. There comes a point however, when the AU window is large enough to compensate for the extra parallelism on the DU, and more address operations can be pipelined to hide the latency.

In the short to medium term high performance architectures will have window sizes in the range that shows a reduction in the latency hiding effectiveness. During this time frame designers will have to incorporate other latency hiding techniques to improve the effectiveness of decoupled machines.

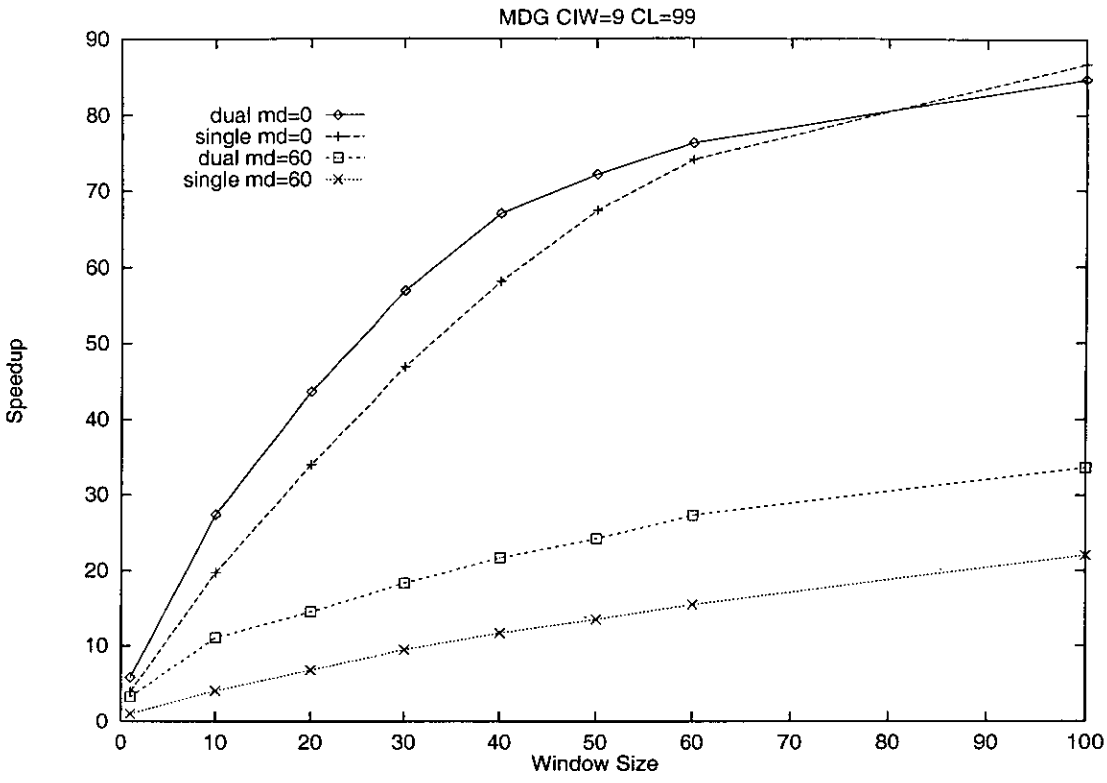


Figure 8.4: Speedup of MDG for varying window size

8.5 Reordering scope and instruction window

The reordering scope and instruction window impose constraints on the degree of reordering in the instruction streams. The reordering scope which is explained in Chapter 7, is used to model different levels of compiler complexity by introducing barriers into the instruction streams. An operation is not able to begin execution until after all operations before the last barrier have executed. The reordering scope can conceptually be thought of as a software window within which operations are permitted to reorder.

Determining reordering scope and window sizes that give equivalent performance allows comparisons to be made between different levels of compiler and hardware complexity. Table 8.5 and 7.2 show the average IPC values for the instruction window and reordering scope, respectively. The results show that on average a window size of 20 to 30 instructions [6] is equivalent to the reordering operations within leaf loops (RS1 and RS2) when the memory system is perfect (md=0). To reorder operations within nested loop boundaries (RS3 and RS4)

[6]The is approximately equal to the window size of the R10000 (window size = 32 instructions) [91]

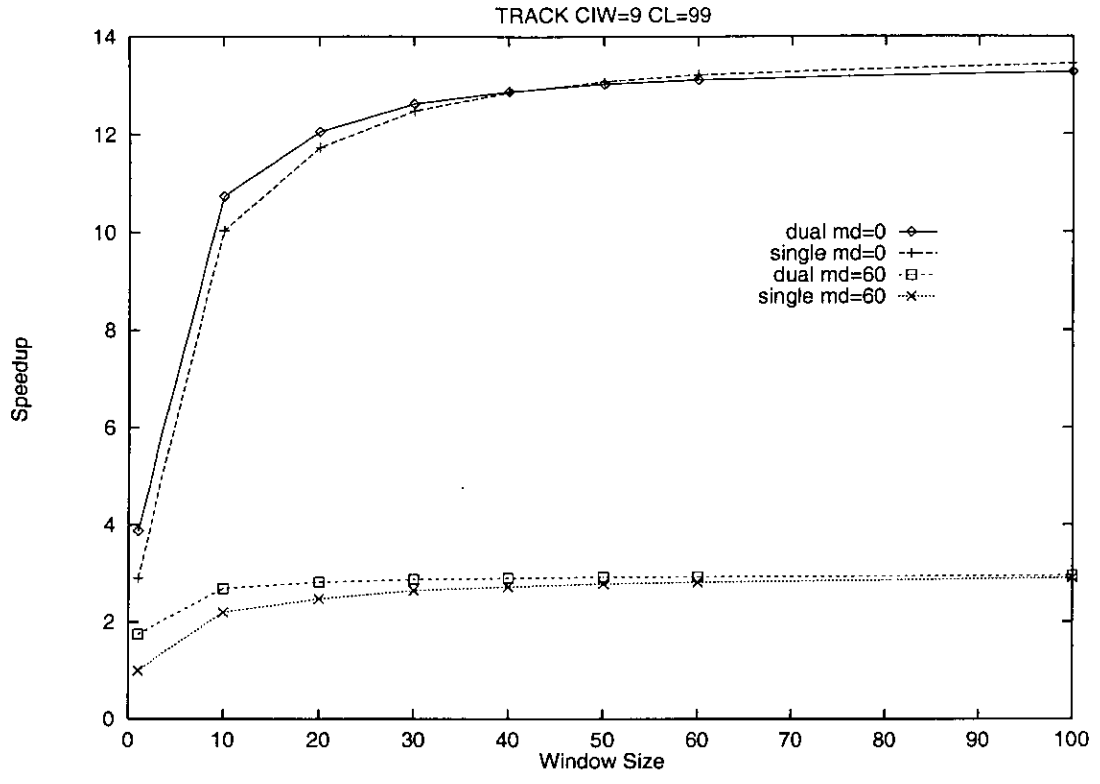


Figure 8.5: Speedup of TRACK for varying window size

requires on average windows of 40 to 50 instructions. However, once the memory differential becomes large (md=60) instruction windows have to be larger than 100 instructions to give the same performance as reordering within nested loops (RS3 and RS4); a window size of 100 instructions has an IPC 83% of the value for RS3.

The conclusion to be drawn from this comparison is that very large instruction windows will be required to achieve the latency hiding effectiveness latent within nested loops. This suggests that while reordering hardware is sufficient for small memory latencies, compiler techniques capable of inter-loop scheduling and analysis are necessary for large memory latencies. The reordering hardware is unable to provide a sufficient degree of reordering to detect the parallelism necessary to tolerate very large memory latencies. To provide a higher degree of reordering would require very large instruction windows that would certainly restrict clock speeds. This result applies in the case of perfect branch prediction for loop closing branches.

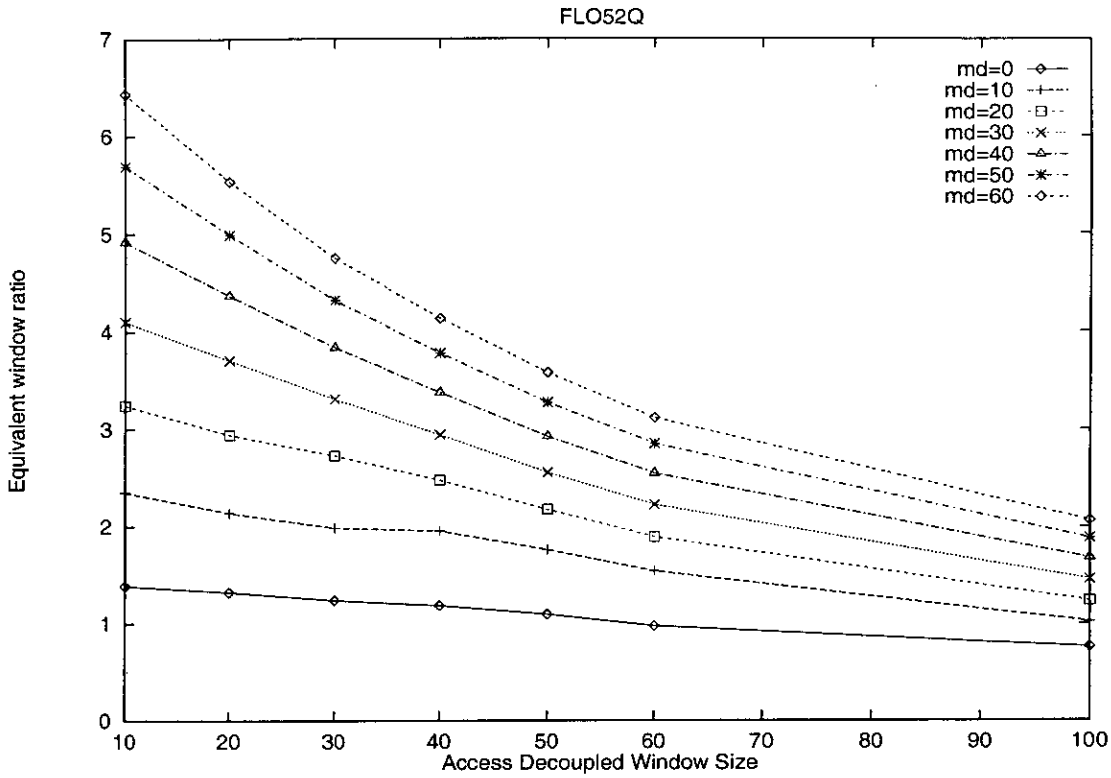


Figure 8.6: Equivalent window ratio for FLO52Q

8.6 Summary

This chapter has focused on two objectives in the design space of future microprocessors; the need to hide large memory latencies and the need to reduce the complexity of instruction issue logic. It has investigated the behaviour of data prefetching on a decoupled machine and a single stream, out-of-order superscalar architecture. It has examined the relationship between memory latency, window size and speedup for the two architectures. In order to remove the impact of other architectural issues the experiments have assumed an ideal environment. This environment provides good conditions for data prefetching, high levels of ILP and places the greatest pressure on the latency hiding mechanism.

It has been shown that when window sizes are unlimited, the single stream system has a higher IPC than the dual stream system. However, the results show that at large combined issue widths (9 issue slots), the dual stream system's IPC is only 5-6% lower than the single stream system. At small combined issue widths (2 instruction slots), the dual stream system's IPC is 11-13% lower than the single stream system.

An analytical model has been used to estimate the average window size of the

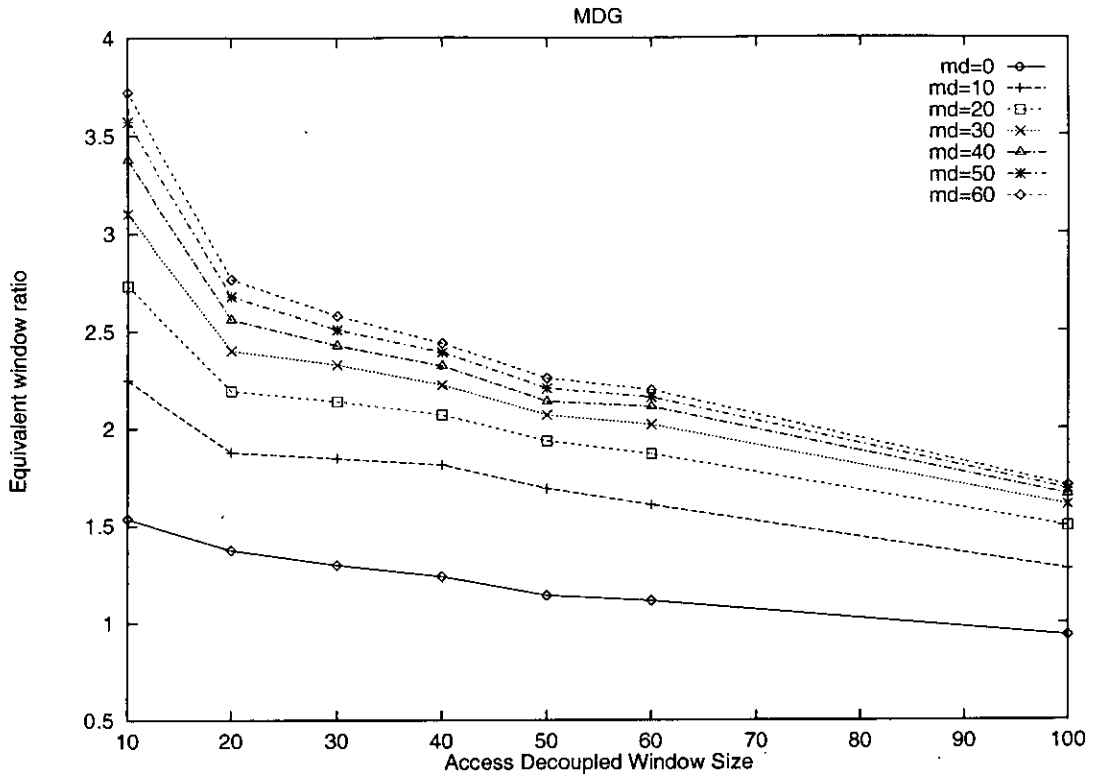


Figure 8.7: Equivalent window ratio for MDG

dual and single stream system. The results have shown that the single stream system consistently has a larger average window size than the dual stream system. It has been shown that the dual stream system is more effective at hiding memory latency than the single stream architecture. For large memory differentials (60 cycles) it has been shown that even for large window sizes of 100 instructions, the dual stream system consistently performs better than the single stream. The results have also shown that to achieve the same speedup as a dual stream architecture the single stream system needs a window size between 2.5 to 5 larger. The increase in window size required to achieve equivalent performance on the single stream system was also found to increase with larger latencies.

It has been shown in [35] that architectures with p processing elements each with a window size N can achieve lower but comparable IPC to a machine with window size pN. This work argued that the simpler hardware would allow faster clock speeds, and hence reduce execution times. However, the results from this chapter allow a stronger claim to made, that the dual stream system can produce higher IPC values than a single stream architecture with window size 2N.

The concept of the effective single window has been introduced to help interpret the experiment results. The effective single window conceptually illustrates

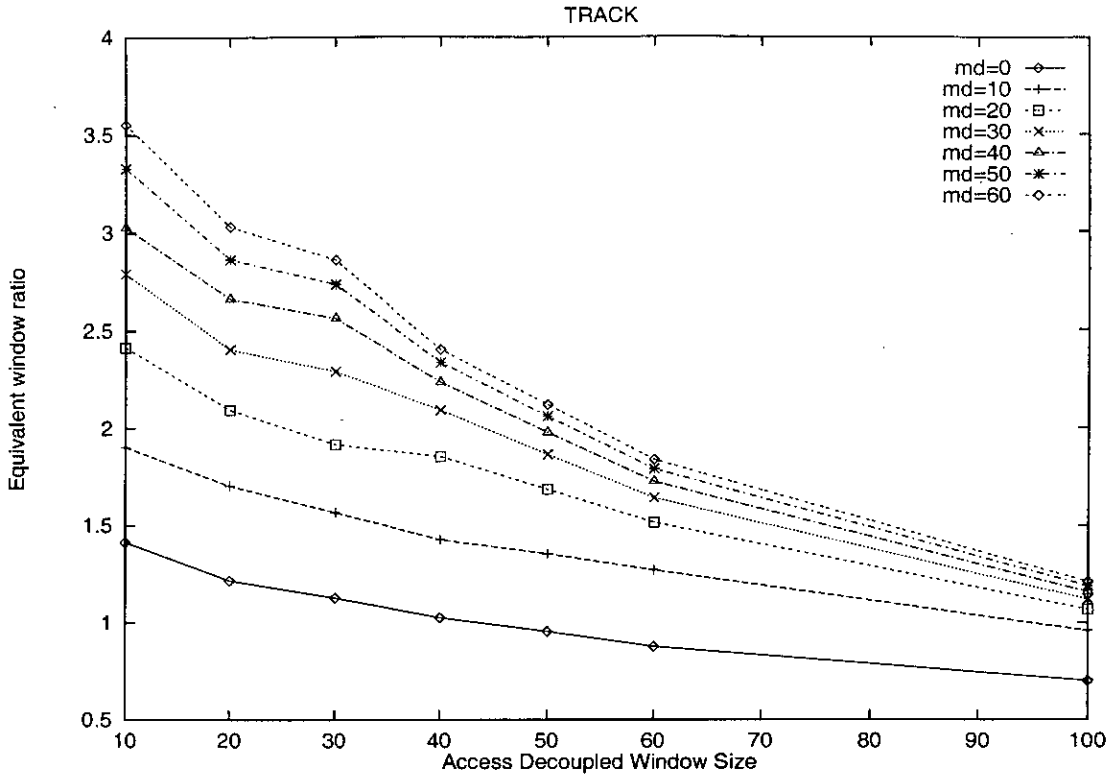


Figure 8.8: Equivalent window ratio for TRACK

how the dual stream system is able to perform better than an single stream system with twice the size of instruction window.

The results have also shown how the latency hiding effectiveness of the dual stream system decreases as the window size increases to 50 instructions. Though the speedup did increase with larger window size the dual stream system was not found to be as effective at hiding latency. However when windows were greater than 50 instructions the LHE was found to improve. This behaviour illustrates the tensions that exist between higher ILP and the access decoupling mechanism.

This chapter has shown that access decoupling can combine the benefits of latency hiding with simplifying the window logic complexity. However, these conclusions have been drawn with the caveat, that the experiments were conducted under conditions that did not restrict decoupling. The next chapter considers the sensitivity of this chapters findings to design issues *unique* to decoupling.

WS	Program	MD=0	MD=60		Program	MD=0	MD=60	
		IPC	IPC	LHE (%)		IPC	IPC	LHE (%)
1	ADM	0.6	0.4	72	DYFESM	0.6	0.4	81
10		2.5	1.3	51		2.5	1.5	59
20		3.7	1.6	43		3.5	1.9	53
30		4.4	1.8	41		4.3	2.1	49
40		4.9	1.9	39		4.6	2.2	48
50		5.3	2.0	38		4.8	2.4	50
60		5.4	2.1	38		5.0	2.4	49
100		5.9	2.3	39		5.3	2.6	49
∞		8.8	8.8	100		7.6	6.6	87
1	FLO52Q	0.6	0.5	82	MDG	0.6	0.3	56
10		2.7	2.1	79		2.6	1.1	41
20		4.5	3.3	74		4.2	1.4	33
30		5.8	4.2	72		5.5	1.8	32
40		6.8	4.8	70		6.4	2.1	32
50		7.3	5.1	70		6.9	2.3	34
60		7.5	5.3	71		7.3	2.6	36
100		8.0	5.8	73		8.1	3.2	40
∞		8.6	8.4	97		8.8	4.3	48
1	QCD2	0.6	0.4	76	TRACK	0.5	0.3	45
10		2.7	1.5	54		1.5	0.4	25
20		4.2	1.9	46		1.7	0.4	23
30		4.9	2.2	44		1.8	0.4	23
40		5.3	2.3	43		1.8	0.4	23
50		5.5	2.4	43		1.8	0.4	22
60		5.6	2.4	43		1.8	0.4	22
100		5.7	2.5	44		1.9	0.4	22
∞		8.1	4.5	55		1.9	0.4	22
1	TRFD	0.6	0.5	90	Average	0.6	0.4	
10		3.1	2.2	72		2.5	1.4	
20		5.4	2.9	53		3.9	1.9	
30		7.4	3.2	43		4.9	2.2	
40		8.0	3.3	41		5.4	2.4	
50		8.0	3.4	42		5.7	2.6	
60		8.0	3.4	43		5.8	2.7	
100		8.1	3.7	45		6.1	2.9	
∞		8.2	8.2	100		7.4	5.9	

Table 8.5: IPC and LHE for varying window size (WS)

Chapter 9

Reducing Hardware Complexity: a Sensitivity Analysis

In Chapter 8 it was shown that decoupling could achieve comparable performance with a window size 2.5 to 5 times smaller than a single stream architecture. However, this assumed conditions that did not restrict decoupling. Critical design issues such as decoupled memory capacity, code expansion, inter-unit communication bandwidth, and access ordering were relaxed to determine upper bounds on performance. This chapter presents results from a series of studies that evaluate the conclusions from Chapter 8 under more realistic conditions. Decoupling introduces the additional complexity of partitioning the code into separate streams. There is no known published work comparing the benefits of static and dynamic partitioning algorithms [1]. This chapter examines, for the first time, the effectiveness of static and dynamic partitioning schemes on a decoupled architecture.

This chapter is structured in the following way. Section 9.1 presents the experimental findings and Section 9.2 concludes with a discussion of its findings.

9.1 Experimental results

Chapter 8 assumed conditions that favoured decoupling; inter unit communication cost was free, and by executing the same number of operations on both the dual and single stream system the effect of code expansion was ignored. It was shown that the effect of code expansion was small, but only where memory latencies and issue widths were large. However, this result was based on modelling a dual and single stream architecture with unlimited window size.

This chapter explores some of the design issues particular to the dual stream architecture under more realistic constraints. Tables 9.1 and 9.2 show the different

[1] Tyson [81] investigated the effectiveness of different static partitioning algorithms, but did not consider dynamic partitioning.

configurations of the dual and single stream systems used in the experiments. Table 9.1 shows the base line architectural configuration. The column labelled sequential shows the architectural configuration used for pure sequential execution of the program. Table 9.2 shows the range of parameters investigated as part of the sensitivity analysis in this chapter.

Table 9.1 and 9.2 shows that some components have unlimited capacity. Relaxing constraints on these resources has the benefit on removing 2nd order effects from the experimental analysis, but could be criticised for being too idealistic. However, in a comparative study, relaxing selective resource constraints benefits both architectures equally and therefore does not prejudice either architecture.

The studies in this chapter consider memory differentials in the range between 0 to 60 cycles. This range was selected because it represents a spectrum of different types of memory system. The low end of the range, 5 to 10 cycles, effectively represents processors integrated with a multilevel cache. The high end of the range, 60 cycles, models processors that interface directly with the main memory. Memory differentials of 0 cycles yield performance upper bounds.

Architecture components	Base		
	sequential	dual	single
Floating operation latency (cycles)	5	5	5
Integer/Address op. latency (cycles)	1	1	1
Function Units	1	{4,5}	9
Issue Width	1	{4,5}	9
Register File	∞	∞	∞
Window size	1	30	30
Comm. bus latency	N/A	0	N/A
Comm. bus width	N/A	∞	N/A
Prefetch buffer	N/A	N/A	∞
Decoupled Memory	N/A	∞	N/A
Memory/Buffer bandwidth	∞	∞	∞
Buffer access latency (cycles)	1	1	1
Memory differential (cycles)	60	60	60
Memory Ordering	N/A	wko	wko
Dependency analysis	perfect	perfect	perfect
Code Partition	N/A	static	N/A

Table 9.1: Base-line : architectural configurations for the dual and single stream systems

9.1.1 The effect of load data buffer capacity

In Chapter 8 the decoupled memory and prefetch buffer were assumed to be of unlimited size. However, this assumption could be criticised on the grounds that it unfairly benefits the dual stream system, allowing it to decouple to an unrealistic

Architecture components	Other Values Considered	
	dual	single
Window size	{20,30,40}	{20,30,40}
Comm. bus latency	{0,1,2,4,6}	N/A
Comm. bus width	{1,2,∞}	N/A
Load data buffer capacity	{1, 10, 20, 30, 60, 90, 120, 180, 240, 300}	N/A
Memory differential (cycles)	{0,5,10,20,60}	{0,5,10,20,60}
Memory Ordering	{sto, sso, wko}	{sto, sso, wko}
Code Partition	{static,dynamic}	N/A

Table 9.2: Sensitivity analysis : architectural configurations for the dual and single stream systems

degree. This section describes a set of measurements that were made of the occupancy of the decoupled memory and the prefetch buffer. The occupancy is defined in terms of the number of concurrent active lifetimes in the decoupled memory in a given cycle; where a lifetime is defined:

- for a decoupled load, as the number of cycles between when an AU sends an address to the decoupled memory and the DU fetches the result.
- for a decoupled store, as the number of cycles between the store address being sent to the decoupled memory and it being matched with the corresponding definition.

Table 9.3 shows for the decoupled memory and prefetch buffer, the occupancy as a percentage of the total execution time. For example, in ADM the occupancy was less than 291, 330 and 1,307 for 90%, 95% and 99%, respectively, of the total execution time of the program. Whereas in TRACK, the occupancy of the decoupled memory was less than 113 for 99% of the entire execution of the program. This result is to be expected, as programs with higher levels of parallelism will have more inflight accesses than programs like TRFD.

Table 9.3 shows a striking difference between the occupancy of the decoupled memory and prefetch buffer. It shows, that for the majority of programs the occupancy of the decoupled memory is far in excess of the prefetch buffer. For example, the column labelled 99% shows that for ADM the occupancy of the decoupled memory is approximately a factor of 25 times larger than prefetch buffer. This result is most apparent in highly parallel programs like TRFD. Highly parallel programs have few losses of decouplings, therefore, in the absence of any resource constraints, the number of in-flight accesses increases to very large values.

This finding clearly supports the criticism that relaxing the decoupled memory and prefetch buffer size has skewed the experiments in favour of the dual stream system. However, this result does not indicate the decoupled memory capacity actually required to achieve optimal or near optimal performance.

The degree of decoupling between the AU and DU is mainly limited by the number of loads that can be in-flight at any one time [2]. This number is constrained by the size of the load data buffer, described in Section 7.4. The load data buffer is therefore the critical resource in the decoupled memory, that determines the degree of decoupling in a dual stream system

Figure 9.1 and Table 9.4 show the effectiveness of the dual stream system for a range of load data buffer capacities. The effectiveness is measured by $T(\infty)/T(n)$, where $T(\infty)$ and $T(n)$ are the times to execute the program with a load data buffer of unlimited size and of size n, respectively. Table 9.4 clearly shows, for all programs, that a dual stream system with a load data buffer capacity of 120 can achieve greater than 97% of the optimal performance. This is an encouraging result, as it demonstrates that we don't need to build very large decoupled memories for the dual stream system to outperform the single stream system.

Program	Percentage of Cycles					
	Decoupled Memory			Prefetch Buffer		
	90 %	95%	99%	90%	95%	99%
ADM	291	330	1,307	23	26	54
DYFESM	2,175	3,109	4,946	26	28	32
MDG	212	263	326	23	29	37
QCD2	139	250	603	43	48	57
TRACK	33	59	113	16	21	43
TRFD	35,260	44,252	64,971	22	24	33

Table 9.3: Occupancy of the decoupled memory and prefetch buffer

9.1.2 The effect of code expansion

This study uses the base line architecture to determine the effect of code expansion and to compare them with the findings in Chapter 8. The speedup measurements were made over the sequential architecture shown in Table 9.1. The size of the code expansion introduced by the OCTAVE compiler is shown in Table A.1.

Figures 9.2 to 9.4 show, for three benchmark programs, the graph of superscalar speedup when the memory differential (MD) is 5 and 60 cycles. These

[2] In the dual stream system stores only take a single cycle and therefore will not contribute to the degree of decoupling

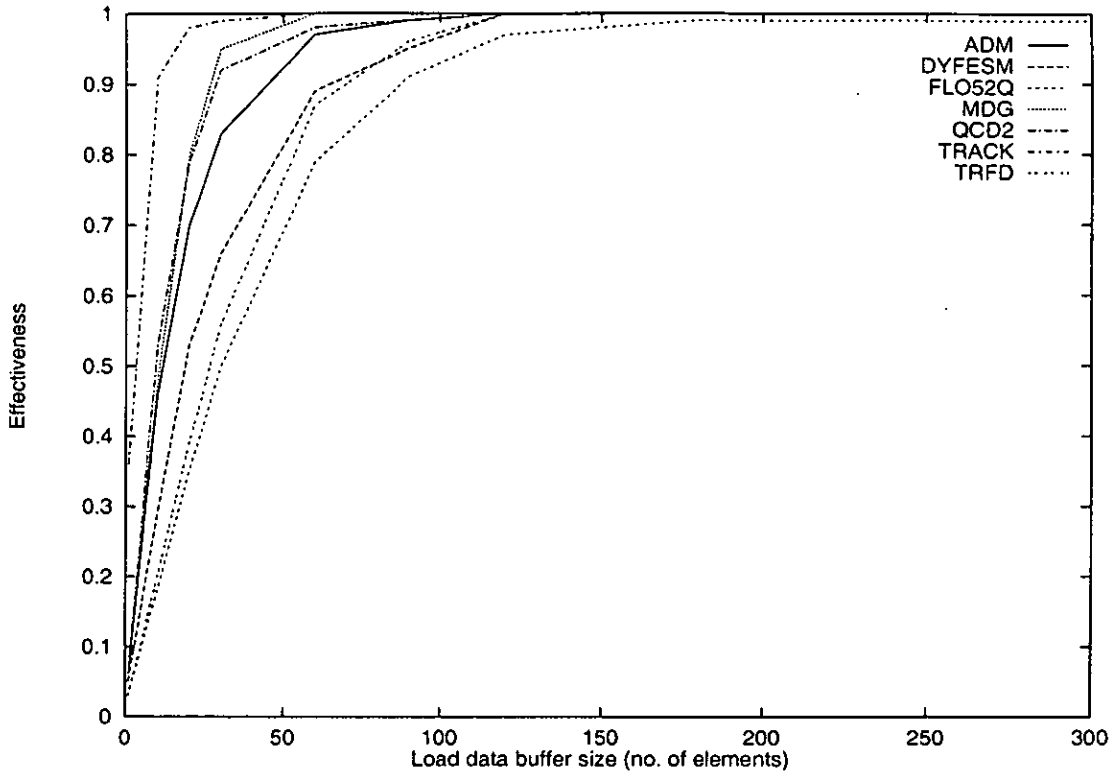


Figure 9.1: Effectiveness of various load data buffer sizes

three benchmark programs, FLO52Q, MDG and TRFD were selected to illustrate, respectively, the behaviour of programs with high, moderate and low levels of parallelism. The graphs show the law of diminishing returns with larger window sizes providing lower performance gains. The horizontal lines show the speedup of the dual stream system for a window size of 20, 30 and 40 instructions (the window size is shown on each line). The single window size that produces equivalent performance to the dual stream system can be derived by projecting down to the x-axis from the point of intersection. The ratio of the the single stream and dual stream window sizes that yield equivalent performance is referred to as

Program	Load Data Buffer Capacity (number of entries)									
	1	10	20	30	60	90	120	180	240	300
ADM	0.06	0.46	0.70	0.83	0.97	0.99	1.00	1.00	1.00	1.00
DYFESM	0.05	0.29	0.53	0.66	0.89	0.95	1.00	1.00	1.00	1.00
FLO52Q	0.03	0.18	0.35	0.50	0.79	0.91	0.97	0.99	0.99	0.99
MDG	0.06	0.48	0.80	0.95	1.00	1.00	1.00	1.00	1.00	1.00
QCD2	0.06	0.53	0.79	0.92	0.98	0.99	1.00	1.00	1.00	1.00
TRACK	0.36	0.91	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
TRFD	0.03	0.20	0.39	0.56	0.87	0.96	1.00	1.00	1.00	1.00

Table 9.4: Effectiveness of various load data buffer capacities

the equivalent window ratio. As expected, for architectures with the same window size [3] the dual stream architecture outperforms the single stream system.

Table 9.5 shows equivalent window ratio for a range of memory differentials. The dual stream window size is shown on the column label. The results show that on average, when memory differentials are 5 cycles, a dual stream system with window of size 20 would achieve similar performance to a single stream architecture with a window size 1.6 times larger. At a large memory differential of 60 cycles the window would have to be 4.2 times larger. Given the quadratic relationship between issue logic complexity and window size the dual stream system offers the benefits of higher performance through faster clock speed and greater reordering.

Table 9.5 also shows that for all programs the equivalent window ratio increases with larger memory differentials. Larger latencies allow greater slippage between the AU and DU providing a greater degree of reordering between the two instruction windows. Another emergent trend is that larger window sizes result in smaller equivalent window ratios. Large single stream window sizes provide opportunities to utilise the full issue width and reorder operations to a similar degree as the dynamic slippage on the dual stream system.

Table 9.6 shows the equivalent single window ratio when the single stream system executes the same number of operations as the dual stream architecture [4]. Comparing the results with Table 9.5 it can be seen that there is minimal difference between the equivalent single window ratio. This leads us to conclude that the effect of the code expansion introduced by the OCTAVE compiler has little effect on performance.

9.1.3 Effective communication bandwidth

This section quantifies the effect of inter-unit communication on the findings of the previous section. Using the base line dual stream architecture, this section explores issues of bus latency and width between the two units. The results from this study are shown in Table 9.7. It shows the computed values of equivalent single window ratio. The column labels show the memory differential.

It can be seen that in the majority of cases the dual stream system still performs better than the single stream architecture. However there are cases such as MDG, QCD2 and TRFD where the single stream system performs better than the dual stream architecture. The average values show that for a bus width of

[3] The AU and DU have the same size of window as the single stream system

[4] This was the case in Chapter 8, where operations are duplicated on the AU and DU in the dual stream system, two operations are also executed on the single stream architecture.

1 word the single stream system outperforms the dual stream architecture when the bus latency is between 2 to 6 cycles and the memory differential is between 0 to 5 cycles. The reason for this is due to the tradeoff between dynamic reordering and communication overhead. The large overhead incurred by inter-unit communication can not be offset by the benefits of dynamic reordering between AU and DU instruction windows; small memory differentials limit the degree of inter-window reordering. However, when the memory differential becomes greater than 5 cycles the degree of reordering increases and the dual stream system performs better than the single stream system.

As expected, Table 9.7 also shows that the range in which dual stream architecture outperforms the single stream system can be extended by increasing the bus width to 2 words. For example, in QCD2, when the memory differential is 5 cycles, a dual stream system only outperforms a single stream architecture when the bus latency is 1-2 cycles. However, when the bus width is increased to 2 words, this range is increased to 1-4 cycles.

The columns labelled 30 in Table 9.5 effectively show the upper-bound on the equivalent single window ratio. Comparing these results with Table 9.7 it can be seen that, for a bus width of 1 word, near optimal results are achieved for a range of bus latencies from 1 and 2 cycles. For a bus width of 2 words that range increases from 1 to 4 cycles. It can be anticipated that with technology permitting multiple processors per chip the communication cost will be between 1 to 4 cycles. In this range it can be seen that on average the effectiveness of the dual stream system will be unaffected when the bus width is 2 words. For a lower communication latency of 2 cycles, a bus width of one word would be sufficient.

9.1.4 The effect of memory reordering

Previous implementations of decoupling used a semi-strong ordering of memory operations [24]. This section examines the effectiveness of this scheme by comparing it with strong and weak ordering schemes. These experiments used the base line architectures for both machines. The dual stream system had a communication bus width and latency of 1 word and 2 cycles respectively. The results are shown in Table 9.8; column labels show the memory differential.

The major finding is that the semi-strong ordering significantly reduces the benefits of the dual stream architecture. For a memory differential of 10 cycles the speedup of weak ordering is 44% higher than semi-strong ordering scheme. The semi-strong scheme only proves to be noticeably more effective, than the strong scheme, at high memory differentials of 60 cycles. Memory differentials

of 5 to 20 cycles would be typical of future high performance memory systems. In this range the dual stream system offers minimal improvement over the single stream architecture for the STO and SSO schemes.

Comparing the weak and semi-strong ordering schemes we observe three emergent types of program behaviour. In highly parallel programs like ADM, TRFD and FLO52Q the speedup under weak ordering is significantly greater than the semi-strong scheme. The average speedup of these programs for a memory differential of 10 cycles is 1.67 and 1.03 for the weak and semi-strong ordering, respectively; representing an improvement of 62%. The moderately parallel programs, MDG and QCD2, for the same memory differential have an average speedup of 1.22 and 0.99, respectively; representing an improvement of 23%. For TRACK, with little parallelism, the difference between the schemes is negligible.

There is also clearly no benefit to using decoupling with a strong ordering scheme.

9.1.5 Dynamic versus static code partitioning

This section compares the effectiveness of a static and dynamic code partitioning algorithms on the dual stream architecture. The static partitioning is performed by the OCTAVE compiler. Dynamic partitioning is implemented at run-time by splitting the code according to the operation type. Integer and address computations are executed on the AU whilst floating point operations are executed on the DU. The benefit of the static partitioning algorithm is that it tries to reduce inter-unit communication at the cost of increasing code size; where a result is required on both units the compiler may duplicate an operation. Code placement is not based on operation or data type, but on a bi-directional data flow algorithm that partitions in ways that would be impossible without knowledge of "future" instruction sequences. The benefit of dynamic partitioning is that it reduces code expansion [5] but increases the communication traffic between units. Dynamic partitioning was used in architectures like the ZS-1 [24].

These experiments used the base line dual stream architecture with a bus latency and width of 4 cycles and 1 word, respectively. Figure 9.5 shows the speedup of the static over the dynamic schemes. The columns are labelled with the memory differential.

In general, as memory differential increases, the performance advantage of static over dynamic partitioning decreases and tends to converge for large memory differentials. On average the speedup decreases from 1.16 to 1.08 for a memory

[5] Some expansion still occurs because of operations for communicating results between units

differential of 5 to 60 cycles. This is due to the cost of transfers between units becoming relatively cheaper at larger memory differentials. The benefits of lower inter unit communication, provided by static partitioning, are reduced and the effect of the code expansion becomes more significant. This can be witnessed by the negative gradient of most of the graphs in Figure 9.5.

The major exception is MDG where dynamic partitioning performs better than static partitioning. This is due to the static scheme increasing the communication traffic between units; the number of AU to DU and DU to AU transfers increased by 11% and 31%, respectively. MDG is a program that requires sophisticated compiler techniques to extract it's moderate levels of parallelism [10]. The OCTAVE compiler only performs intra-procedural analysis when partitioning the code. This limitation may explain the poor code placement of the static scheme in this case.

QCD2 shows that even with large code expansion (15%) the static scheme significantly improves the performance of the dual stream system. The large volumes of communication traffic between units is the performance bottleneck for the dynamic scheme. There are respectively, 2.7 and 4.2 times more transfers from AU to DU and DU to AU using the dynamic scheme.

9.1.6 Speedup and latency hiding effectiveness

This section compares the speedup and latency hiding effectiveness of the dual and single stream system. The latency hiding effectiveness is a measure of the sensitivity of the architecture to increases in the memory latency.

These experiments were made with the base line architectures, with the dual stream system having a communication bus latency and width of 2 cycles and 1 word respectively. To make reasonable comparisons between similar architectural configurations, the single stream system used a window equal in size to the combined AU and DU window sizes i.e. 60 instructions. The experiments also considered the effect of the including a bypass mechanism (see Chapter 3) in the decoupled memory and prefetch buffer. The speedup measurements were made over the sequential architecture shown in Table 9.1.

Figure 9.6 shows the average speedup as a function of memory differential. In the key, the yes and no denotes if the architecture includes the bypass mechanism. It can be seen that the dual stream system is less sensitive to increases in the memory differential and therefore more effective at hiding latency. At low memory latencies (0 to 10 cycles) the single stream system has a larger speedup, but its weaker latency hiding leads to poorer performance when memory differentials are

greater than 10 cycles.

At large memory latencies the bypass mechanism improves the effectiveness of both the dual and single stream architectures. However, the dual stream system profits far more from this optimisation than the single stream architecture. For a memory differential of 60 cycles the dual and single stream average speedups increase by 69% and 11%, respectively. The bypass mechanism exploits the temporal locality exposed by reordering operations to hide memory latencies and reduce memory traffic. The dual stream system can reorder operations to a greater degree, and is therefore able to uncover more temporal locality than the single stream architecture.

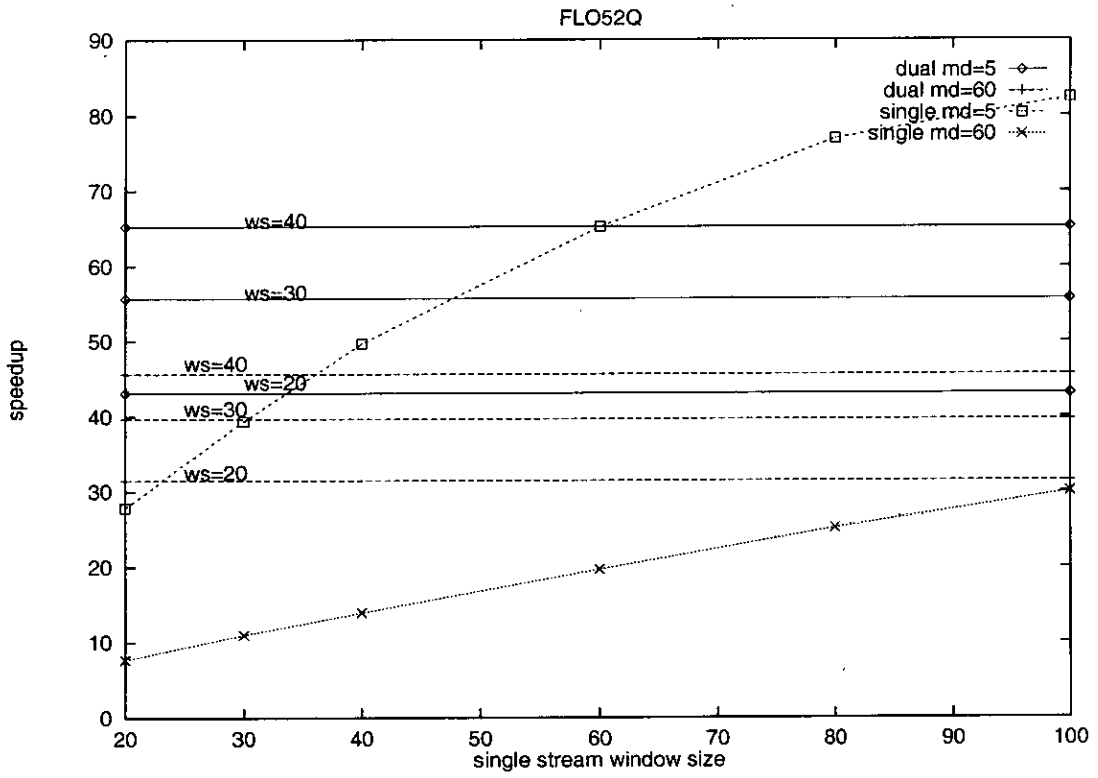


Figure 9.2: FLO52Q speedup

9.2 Summary

This chapter has explored some important regions in the design space of a dual stream decoupled architecture and analysed its effectiveness at reducing instruction issue logic complexity. Comparative studies have been made between a dual and single stream architecture.

One of the significant results from this chapter is that for the dual stream system it is possible to build a load data buffer of a realistic size to achieve near

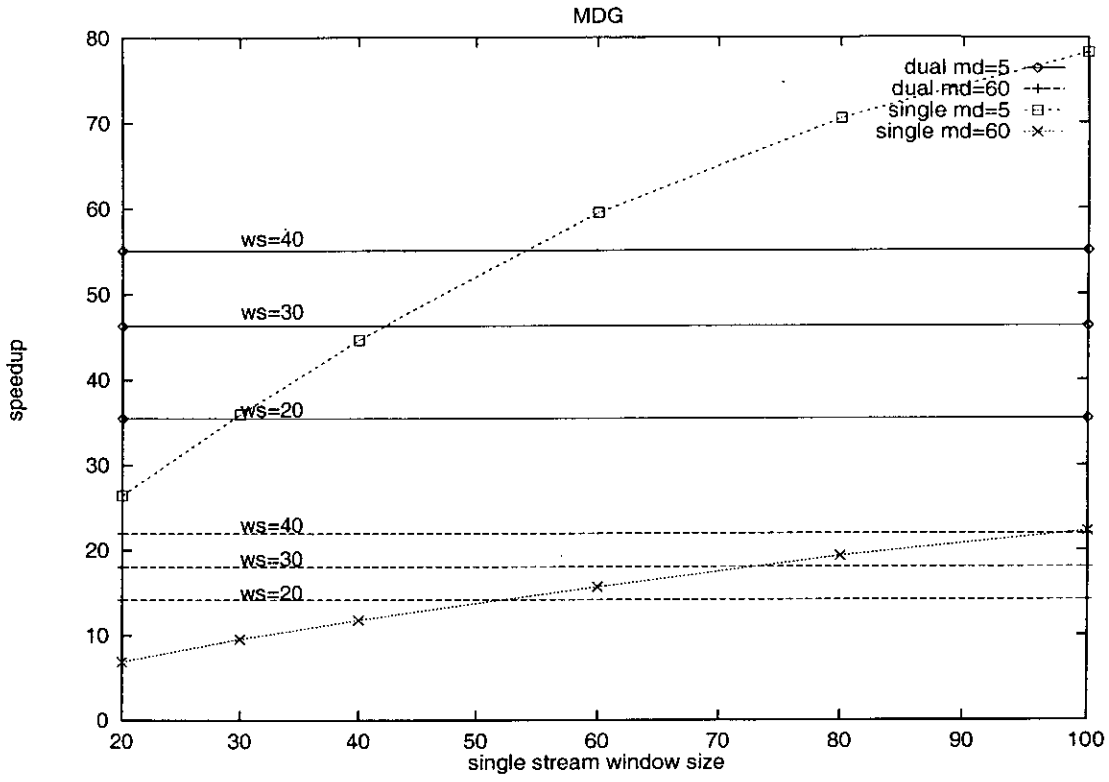


Figure 9.3: MDG speedup

optimal performance. It has been found that a dual stream system with a load data buffer of 120 elements can achieve greater than 97% of the performance of the same architecture with an unlimited buffer.

It has also been shown that when memory differentials are greater than 20 cycles the dual stream architecture performs better than a single stream system of twice the individual window size of a decoupled unit. For memory differentials of between 5 to 20 cycles this factor decreases to between 1.4 to 1.8 times larger, when the communication bus latency is a realistic value of 2 cycles. This factor may actually be larger because savings in reordering logic will reduce delays and allow faster clock speeds.

The results have shown that the code expansion, introduced by the complier, has little influence on the effectiveness of decoupling. A more significant issue is the communication bandwidth between units. It has been found that at small memory differentials (1-5 cycles) and large bus latencies (2 to 6 cycles) between AU and DU, the benefits of decoupling are completely removed. The experiments have identified effective bus width and latencies that achieve near optimal performance.

Memory reordering has been shown to be the most critical design issue for the dual stream architecture. Interestingly, the semi-strong ordering, which is similar

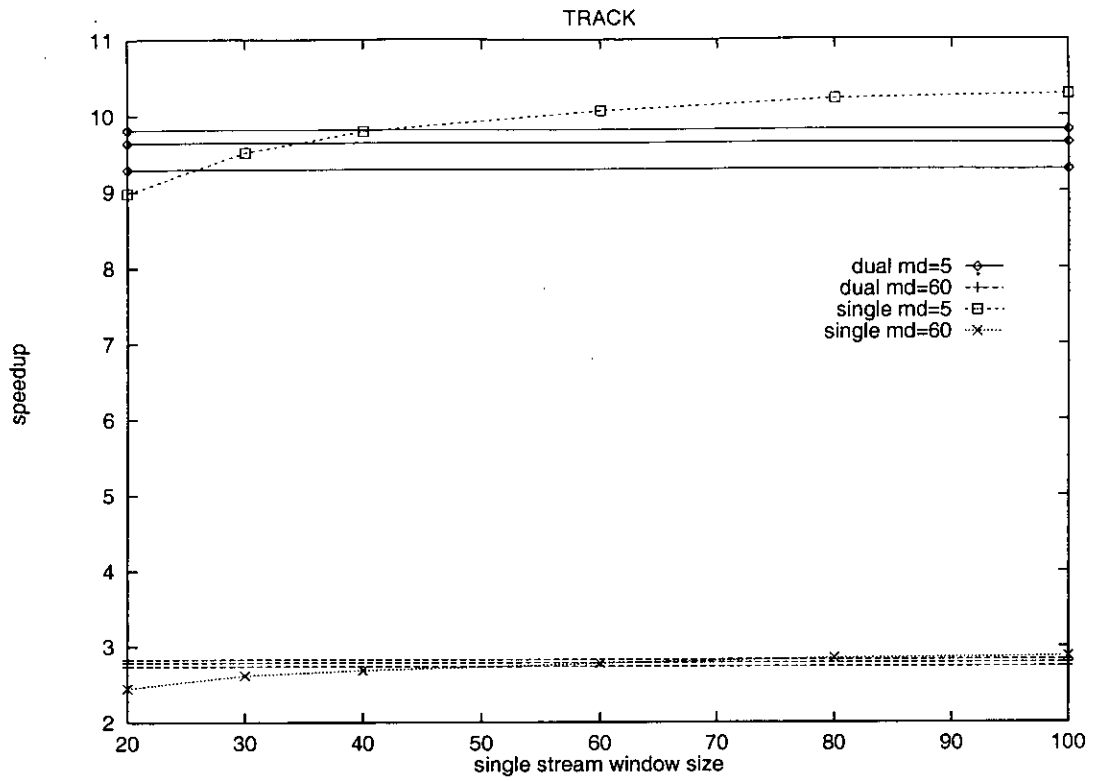


Figure 9.4: TRACK speedup

to a scheme used in previous implementations of decoupling [24], was found to severely limit performance at lower memory latencies.

There has been no known published work that has compared dynamic and static partitioning algorithms for a decoupled machine. It has been shown that in general the static partitioning scheme performs 16% better than a dynamic scheme that partitions by type.

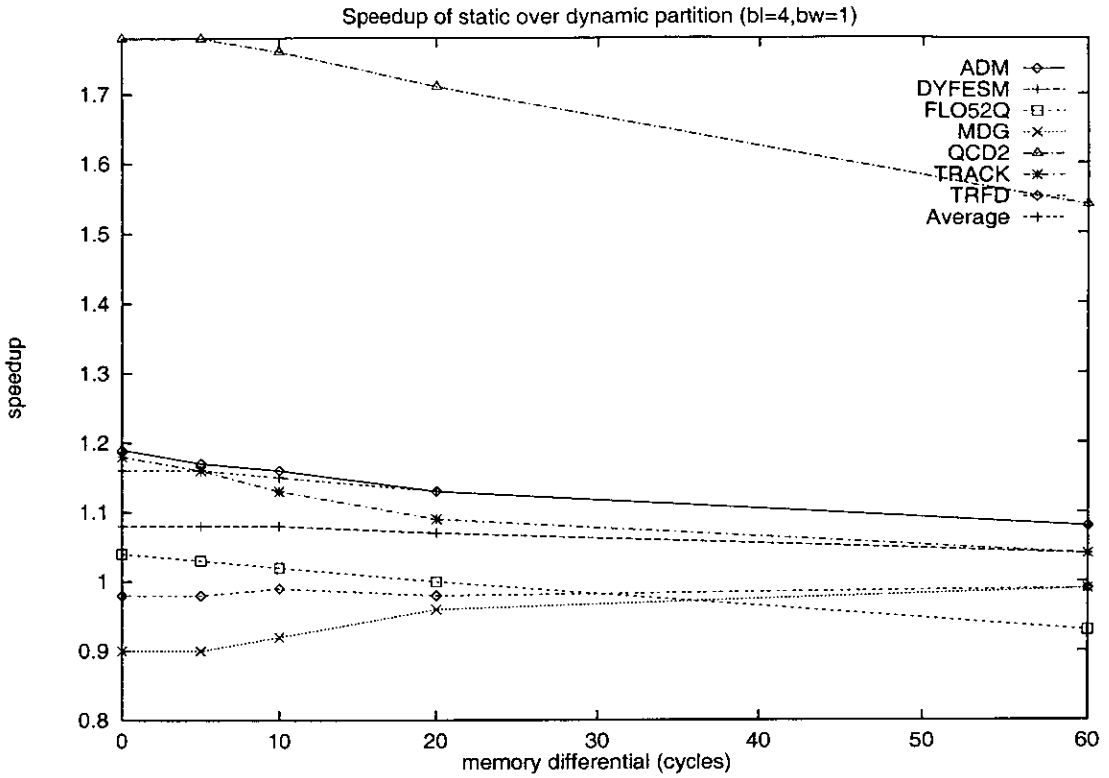


Figure 9.5: Speedup of static over dynamic code partition

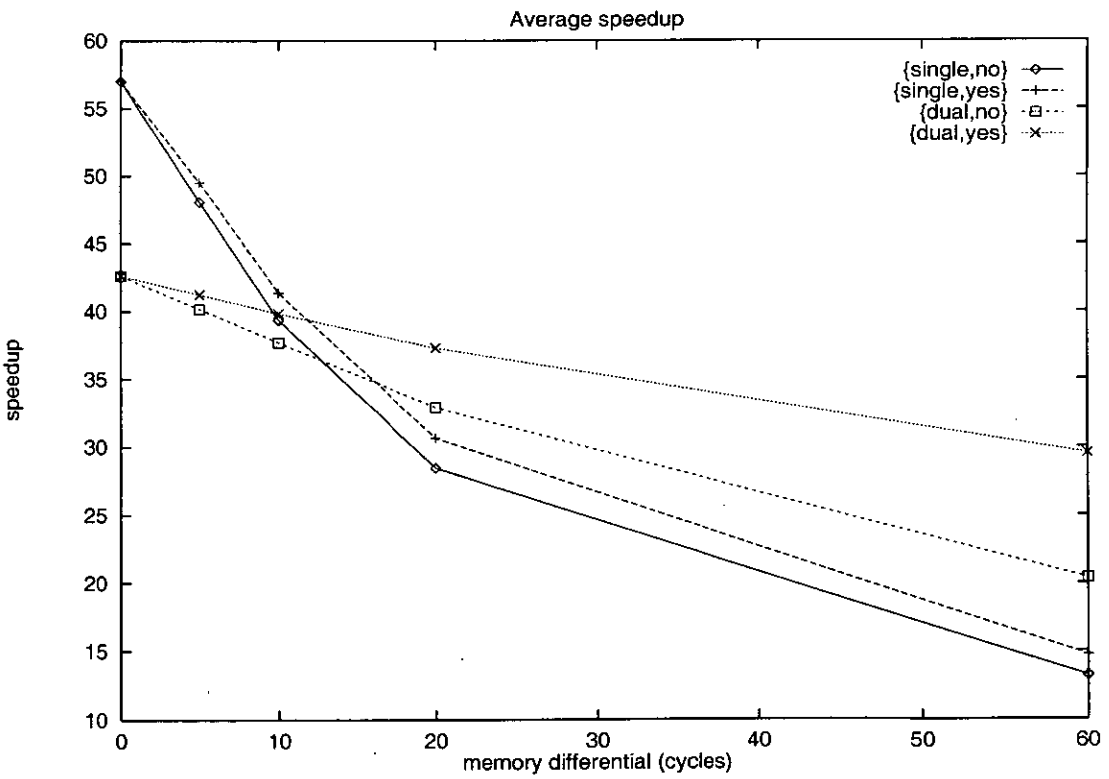


Figure 9.6: Average speedup of baseline dual and single stream systems

program	Equivalent single window ratio								
	md=0			md=5			md=10		
	20	30	40	20	30	40	20	30	40
ADM	1.3	1.2	1.1	1.7	1.6	1.4	2.0	1.9	1.7
DYFESM	1.3	1.2	1.0	1.7	1.6	1.4	2.1	1.9	1.6
FLO52Q	1.3	1.2	1.2	1.7	1.6	1.5	2.1	1.9	1.8
TRFD	1.3	1.2	1.0	2.0	1.9	1.5	2.6	2.5	2.0
MDG	1.2	1.1	1.1	1.5	1.4	1.4	1.7	1.7	1.6
QCD2	1.1	1.0	0.8	1.3	1.1	1.0	1.5	1.3	1.3
TRACK	1.1	1.0	0.9	1.3	1.2	1.0	1.5	1.3	1.2
Average	1.2	1.2	1.0	1.6	1.5	1.3	1.9	1.8	1.6

program	Equivalent single window ratio								
	md=20			md=60					
	20	30	40	20	30	40			
ADM	2.8	2.4	2.2	4.4	3.7	3.1			
DYFESM	2.9	2.6	2.1	4.7	3.5	2.8			
FLO52Q	2.9	2.6	2.4	5.3	4.7	4.1			
TRFD	3.8	3.5	2.8	5.9	4.3	3.4			
MDG	2.1	2.0	1.9	2.6	2.4	2.5			
QCD2	1.9	2.0	2.1	3.8	3.2	2.7			
TRACK	1.8	1.6	1.4	2.6	2.1	1.9			
Average	2.6	2.4	2.1	4.2	3.4	2.9			

Table 9.5: Equivalent single window ratio without code expansion

program	Equivalent single window ratio											
	md=0			md=10			md=20			md=60		
	20	30	40	20	30	40	20	30	40	20	30	40
ADM	1.3	1.3	1.2	2.1	1.9	1.8	2.8	2.6	2.3	4.6	3.8	3.1
DYFESM	1.3	1.2	1.0	2.1	2.0	1.8	2.9	2.7	2.2	4.7	3.6	2.9
FLO52Q	1.3	1.2	1.2	2.1	2.0	2.0	2.9	2.7	2.5	5.5	4.8	4.1
TRFD	1.4	1.3	1.1	2.7	2.7	2.2	3.9	3.5	2.8	5.8	4.3	3.3
MDG	1.4	1.3	1.2	1.9	1.8	1.8	2.2	2.1	2.1	2.8	2.6	2.4
QCD2	1.3	1.1	1.0	1.7	1.5	1.4	2.0	2.2	2.2	3.9	3.3	2.8
TRACK	1.2	1.1	1.0	1.7	1.6	1.4	2.1	1.9	1.9	3.0	2.9	2.4
Average	1.3	1.2	1.1	2.0	1.9	1.8	2.7	2.5	2.2	4.3	3.6	3

Table 9.6: Equivalent single window ratio with code expansion

Program	Bus Latency (cycles)	Equivalent single window ratio									
		Bus width = 1					Bus width = 2				
		0	5	10	20	60	0	5	10	20	60
ADM	1	1.2	1.5	1.9	2.4	3.7	1.2	1.5	1.9	2.4	3.8
	2	1.2	1.5	1.9	2.4	3.7	1.2	1.5	1.9	2.4	3.7
	4	1.1	1.5	1.8	2.4	3.6	1.2	1.5	1.8	2.4	3.7
	6	1.0	1.4	1.8	2.4	3.6	1.1	1.5	1.8	2.4	3.6
DYFESM	1	1.2	1.6	1.9	2.6	3.5	1.2	1.6	1.9	2.6	3.5
	2	1.2	1.6	1.9	2.6	3.5	1.2	1.6	1.9	2.6	3.5
	4	1.2	1.6	1.9	2.6	3.5	1.2	1.6	1.9	2.6	3.5
	6	1.1	1.6	1.9	2.6	3.5	1.1	1.6	1.9	2.6	3.5
FLO52Q	1	1.2	1.6	1.9	2.6	4.6	1.2	1.6	1.9	2.6	4.7
	2	1.2	1.5	1.9	2.5	4.6	1.2	1.6	1.9	2.6	4.7
	4	1.1	1.4	1.8	2.4	4.5	1.2	1.5	1.9	2.5	4.6
	6	1.0	1.3	1.7	2.3	4.3	1.1	1.5	1.8	2.5	4.6
TRFD	1	1.2	1.9	2.5	3.5	4.3	1.2	1.9	2.5	3.5	4.3
	2	1.2	1.9	2.5	3.5	4.3	1.2	1.9	2.5	3.5	4.3
	4	1.2	1.9	2.5	3.5	4.3	1.2	1.9	2.5	3.5	4.3
	6	1.2	1.8	2.5	3.4	4.3	1.2	1.8	2.5	3.4	4.3
MDG	1	1.1	1.4	1.6	1.9	2.4	1.1	1.4	1.6	1.9	2.4
	2	1.0	1.3	1.5	1.9	2.4	1.0	1.3	1.6	1.9	2.4
	4	0.8	1.0	1.3	1.7	2.3	0.9	1.2	1.4	1.8	2.3
	6	0.6	0.8	1.0	1.4	2.2	0.8	1.1	1.3	1.7	2.3
QCD2	1	0.9	1.1	1.3	1.9	3.2	0.9	1.1	1.3	1.9	3.2
	2	0.8	1.0	1.2	1.7	3.2	0.9	1.1	1.3	1.9	3.2
	4	0.7	0.8	1.0	1.3	3.0	0.8	1.0	1.1	1.7	3.2
	6	0.6	0.7	0.8	1.1	2.7	0.7	0.9	1.0	1.4	3.1
TRACK	1	0.9	1.0	1.2	1.4	2.1	0.9	1.1	1.2	1.5	2.1
	2	0.8	0.9	1.1	1.3	2.0	0.8	1.0	1.2	1.4	2.0
	4	0.5	0.8	0.9	1.2	1.9	0.7	0.9	1.0	1.3	1.9
	6	0.3	0.6	0.8	1.0	1.7	0.5	0.8	0.9	1.2	1.9
Average	1	1.1	1.4	1.8	2.3	3.4	1.1	1.5	1.8	2.3	3.4
	2	1.1	1.4	1.7	2.3	3.4	1.1	1.4	1.8	2.3	3.4
	4	0.9	1.3	1.6	2.2	3.3	1.0	1.4	1.7	2.3	3.4
	6	0.8	1.2	1.5	2.0	3.2	0.9	1.3	1.6	2.2	3.3

Table 9.7: Equivalent single window ratio for various bus and memory latencies

Program	Access order scheme	Speedup				
		0	5	10	20	60
ADM	wko	1.09	1.25	1.39	1.60	2.03
	sso	0.96	0.99	1.04	1.15	1.56
	sto	1.00	1.00	1.01	1.01	1.18
DYFESM	wko	1.09	1.30	1.46	1.73	2.32
	sso	1.00	1.01	1.01	1.01	1.42
	sto	1.00	1.01	1.01	1.01	1.01
FLO52Q	wko	1.11	1.38	1.67	2.17	3.60
	sso	0.99	1.03	1.08	1.21	1.95
	sto	1.00	1.00	1.00	1.00	1.07
MDG	wko	0.98	1.19	1.35	1.56	1.85
	sso	0.87	0.92	0.97	1.05	1.30
	sto	1.00	1.00	1.00	1.01	1.02
QCD2	wko	0.90	0.99	1.09	1.25	1.55
	sso	0.89	0.94	1.01	1.10	1.27
	sto	0.91	0.93	0.95	0.97	1.15
TRACK	wko	0.96	0.99	1.01	1.03	1.06
	sso	0.94	0.98	0.99	1.01	1.03
	sto	0.96	0.99	1.00	1.01	1.02
TRFD	wko	1.23	1.73	2.16	2.85	3.44
	sso	1.00	1.00	1.00	1.00	1.78
	sto	1.00	1.00	1.00	1.00	1.01
Average	wko	1.05	1.26	1.44	1.74	2.26
	sso	0.95	0.98	1.01	1.07	1.47
	sto	0.98	0.99	1.00	1.00	1.06

Table 9.8: Speedup of the dual over the single stream system for various access ordering schemes

Chapter 10

Conclusions

This thesis is the first known study into the effectiveness of out-of-order decoupled superscalar architectures in achieving the objective of increasing ILP while reducing issue logic complexity. The effectiveness of decoupling at achieving this objective has been analysed through the perspective of a limitation study. This study has been performed by focusing on those design issues that determine the degree of reordering and relaxing other architectural constraints.

This thesis' narrative has been developed in the following way. Chapters 5 and 6 have provided the groundwork to the thesis, developing an understanding of the behaviour and characteristics of the out-of-order decoupled architectures. Chapters 7, 8 and 9 have built on this work to present the key findings of the thesis.

In Chapter 1 it was postulated that out-of-order decoupled superscalar architectures could, through control and access decoupling provide a solution to the problems caused by large memory latencies, issue logic complexity and control dependencies. This chapter summarises the thesis findings in relation to this original postulate.

During the course of this research it has been necessary not only to review the work in relation to its experimental findings, but also to consider and respond to the technological trends that have taken place during the period. Section 10.1 summarises the main original findings in the thesis and Section 10.2 situates the work in the context of emerging technological trends. Finally, Section 10.3 outlines future work suggested by the thesis.

10.1 Thesis findings

The main findings from this thesis are as follows:

1. For large issue widths an access decoupled machine is less sensitive to increases in memory latency than a single stream out-of-order superscalar architecture.

2. A single stream out-of-order superscalar architecture requires an instruction window 2.5 to 5 times larger than a decoupled machine to achieve equivalent performance.

3. Memory access reordering is the most critical design issue for decoupled architectures. A memory ordering scheme similar to one used in certain previous implementations of decoupling severely limits performance at low memory latencies.

4. Load data buffers in the decoupled memory, used to support out-of-order accesses in the dual stream system, can be built at a realistic size to achieve near optimal performance.

5. The static partitioning algorithm of the OCTAVE compiler performs 16% better than a dynamic scheme that partitions by type, alone.

6. Communication bandwidth has a significant effect on the performance of the decoupled architecture. At low memory latencies the benefits of decoupling are completely removed.

7. The results from points 1 and 2 are only marginally effected by code expansion.

8. For the functional decomposition used in the OCTAVE compiler control decoupling does not sufficiently utilise resources to merit dedicated hardware support. Control decoupling is therefore not considered a practical technique for removing control dependencies with this partitioning algorithm.

9. Balanced configurations of AU and DU that ensure optimal throughput have been identified.

In summary, these findings show that the dual stream system through dynamic reordering within relatively small instruction windows provides one possible solution to the problem of issue logic complexity and latency hiding and represents a viable alternative to a single stream out-of-order superscalar architecture - at least for the applications considered in this thesis. The significant advantage of the dual stream system is that for memory differentials greater than 20 cycles it

can achieve a a higher speedup than a single stream system of twice the individual window size of a decoupled unit. An explanation of this effect has been described through the concept of the Effective Single Window. In the following section, this thesis discusses how these findings relate to existing research.

10.2 Related work

Certain trends and developments during the last few years lend support to the use of decoupling in superscalar architectures. The rest of this section discusses these issues and relates the contributions of this thesis to these developments.

The gap between processor and memory speeds is still growing exponentially which means we could hit the memory wall within the next decade [89]. Intelligent DRAM (IRAM) has been proposed as a way of reducing memory access times [28] by merging the processor and DRAM on a single chip. This has the benefit of increasing memory bandwidth and reducing memory latency. The disadvantage of IRAM is the reduction in speed of the processor logic. However, with it's high bandwidth and low latency IRAM could service the multiple accesses of a large issue width decoupled processor; the high ILP of such an architecture could then compensate for the slower logic speed.

Latency hiding technology is as critical to high performance architectures in 1999 as it was when this research began in 1992. This thesis has shown, through a limitation study (see Chapter 5), that a decoupled architecture is sensitive to large memory latencies, but when combined with an optimised decoupled memory sensitivity is almost completely removed [1]. Kurian, in [55], showed that decoupling was more effective when used with a data cache. However, Kurian's study assumed a non-interleaved memory and did not consider the effects of reordering or issue width on performance.

The use of access decoupling has become increasingly more prevalent in superscalar architectures. For example, a small degree of decoupling can be seen in the use of non-blocking loads for lock-up free caches. Most high performance machines can support between 4 to 8 outstanding loads to a first level cache. A higher degree of decoupling can be seen in the PA-RISC architectures [27]. They have a separate instruction window for memory operation and a dynamic buffer for reordering accesses.

Superscalar architectures are being developed to support larger issue widths and more aggressive operation reordering. However, extracting higher levels of

[1] However, this was in the case where all resource constraints were relaxed and only data dependencies were enforced

ILP increases issue logic complexity introducing delays critical to processor clock speeds. To solve this problem a number of designers have proposed the use of *decentralised issue logic.* Palacharla *et al.* [64] proposed a dual clustered superscalar processor that simplified issue logic by using multiple FIFO queues in each cluster. Centralised steering logic was used to pass decoded and renamed instructions to the clusters. Issue logic was simplified by only operations on the heads of the queues checking for operand availability. The results from this paper showed that a dual clustered architecture, with a window of size n, would suffer a performance degradation of 12% over a architecture with window size $2n$. However, the improvements in clock speed would result in an average performance increase of 16% for the smaller window machine. Clustering has been used in the Alpha 21264 to reduce the number of read ports to the integer register file. However the 21264 still uses a single integer instruction window to reorder operations.

Farkas *et al.* [35] describes a multicluster architecture that reduces cycle time by employing distributed register files, instruction windows and function units. However, the decode and register renaming is still centralised. The architecture is found to consume more processor cycles than a single cluster machine, but the increase is not sufficient to offset the benefits of a faster clock speed. Farkas comments that the multicluster is similar to a decoupled architecture, but requires that inter-unit data transfers be performed in order. This is more conservative than it needs to be. A mechanism to convey the ordering of operations between units could be implemented in a similar way to the scheme described in Section 7.4. Instructions that moved data between units could be tagged with a sequence number and sent to the target unit. The target unit could then use the tag to wakeup the appropriate operation in its private instruction window.

Decentralised issue logic has also been proposed in the multiprocessor architectures like the Multiscalar [75], PEW [51] and MISC [81]. In the Multiscalar architecture [12, 75, 78] an on chip multiprocessor increases ILP by speculatively executing along a single control flow path in a task graph. Tasks are distributed centrally, but the decode, fetch and issue functions are all performed by the individual processors. The Multiscalar architecture supports a common physical register file, decentralised reordering logic, and provides queues for fast data transfer between processing elements.

The PEW architecture [51] uses centralised decode and renaming logic, but task distribution is finer grained than in the Multiscalar architecture. PEW dispatches individual instructions to processors using an heuristic that ensures dependent operations are executed on the same processor. The results confirm

the findings of Palacharla *et al.* [64] that decentralised reordering windows can achieve comparable performance to a single window architecture. This paper does not however keep the number of function units constant in all cases. The author compares an architecture with a 64-instruction window and 8 function units against 2 processors with window size 16 and 8 functions units. This means the latter has a total issue width of 16.

The MISC machine is a 4 processor decoupled architecture which, like the Multiscalar architecture, uses decentralised fetch, decode and renaming logic. ILP extraction and code partitioning are, however, performed statically by the compiler. Each processor is a single issue in-order processing unit that communicates values through the use of queues.

This thesis complements the work on decentralised reordering logic by considering a functional decomposition of the program. Through the functional decomposition, dynamic slippage and latency tolerance mechanisms, a decoupled architecture has been shown to produce *higher* performance than an architecture with an instruction window twice the size. This result extends the work of previous studies [64, 51] which showed a small degradation in performance. The explanation for the additional performance is due to dynamic slippage that allows the decoupled architecture to look further ahead in the instruction stream. This concept has been explained through the *effective single window.*

Tyson's work on the MISC architecture has strong parallels with this thesis. It is therefore necessary to discuss his work in detail to show this thesis' original contribution. Tyson showed that a 4 processor, 1-way in-order issue decoupled architecture outperforms a 4-way in-order issue Alpha 21164. This thesis extends Tyson's findings by showing that a decoupled architecture can outperform a single stream superscalar architecture.

Tyson also showed the importance of memory reordering, but this was in relation to improving the performance of the MISC cache; all memory operations were issued to the cache in program order.

Smaller silicon feature sizes mean that a single chip multiprocessor could be built within the next few years [44, 29]. Inter-processor communication will all be on-chip, reducing the cost of synchronisation and value passing. A single chip decoupled architecture could therefore be possible in the near future, reducing the effect of inter-unit communication on performance. Point (6) describes how important transfer cost was to the performance of the decoupled architecture. A single chip decoupled architecture would have a low transfer cost between units removing one of the limitations to its performance.

Research into cache technology is showing interesting results into their effectiveness at utilising on-chip resources effectively. Huang and Shen [46] have shown that current superscalar architectures have sufficient bandwidth and memory but inefficient use of resources means that over 50% of values loaded into a cache are never used. The need for improved cache management is also discussed by Burger [13]. Burger shows that the efficiency of a cache is less than 20%; where efficiency is measured as the fraction of the cache that holds live data. The paper shows that with a near optimal cache management scheme bandwidth can be greatly reduced even with small amounts of memory (128-512 bytes). This study argues for the use of decoupling as a technique for reducing memory latency and compiler techniques for optimising cache utilisation.

Decoupling could be used as integral part of a cache management scheme. In such a scheme decoupling would provide the lookahead mechanism to allow specialised hardware, situated between the cache and the address unit, to optimise cache performance. Hardware to perform this type of function is already being discussed in the literature. *Access Combining* [86] is a technique used to combine multiple accesses to a single cache line into a single access with multiple offsets. The technique is used in multi-ported caches to provide multiple accesses per cycle. Memory reordering, as shown in [81], can be used to ensure that accesses can be dispersed across the cache banks to reduce conflicts. Another technique, known as *Stream Controllers* [33, 60], reorders accesses to take advantage of the non-uniform DRAM access times.

Having discussed this thesis findings in relation to emerging technological trends this thesis concludes with a discussion of suggested future work.

10.3 Future work

One of the significant findings from this thesis has been that through out-of-order decoupled execution we can achieve higher performance than a single stream stream system with a window size twice the size of one of the decoupled units. This is a stronger claim than has been made in other studies [51, 64] into systems with decentralised issue logic. They have measured small reductions in IPC, but have rightly assumed that this would be compensated for by reductions in issue logic complexity. However, the central requirement for allowing the AU and DU to slip to a large degree is that the decoupled memory be able to support a large number of inflight loads. Section 7.4 proposed a mechanism for supporting this requirement and Section 9.1.1 showed that the size of this buffer did not have to

be excessively large to achieve comparable performance to a load data buffer of unlimited size.

These results have, however been determined without a detailed modelling of the entire decoupled and main memory system. Future work should therefore consider investigating the effects of factors such as bank conflicts, restricted memory bandwidth and variable memory latencies. It can be speculated, that the impact of these factors will be to increase the required size of the load data buffer. It will be important for future work to determine the extent of this increase.

Appendix A

Program Characteristics

This appendix shows a breakdown of the operations executed on the AU and DU for the dual stream system. Table A.1 gives the code expansion over a single stream architecture. It shows the percentage increase and a breakdown of the increase by integer, float and data transfers operations. The average increase is 9.2 % of which on average 55%, 20% and 25% is due to additional integer, float and transfer operations, respectively. Table A.2 shows the percentage breakdown by type and decoupled unit. The value in brackets shows the percentage as a total of all operations executed on the decoupled architecture. Table A.3 shows the absolute count of operations executed in each of the benchmark programs.

Prog.	Total Increase (%)	Breakdown of Expansion		
		Ints. (%)	Flts (%)	Trans. (%)
ADM	7	70	14	16
DYFESM	3	79	11	10
FLO52Q	4	58	23	19
MDG	14	12	15	73
QCD2	15	62	15	23
TRACK	20	8	65	27
TRFD	2	96	0	4

Table A.1: Percentage increase in the number of operations executed by the dual stream system

Prog.	Op Type	AU				DU			
		ops.	loads	stores	total	ops.	loads	stores	total
ADM	flts	3 (1)	2 (1)	1 (0)	6 (3)	40 (23)	26 (15)	23 (13)	89 (50)
	ints	91 (39)	1 (0)	3 (1)	94 (41)	11 (6)	0 (0)	0 (0)	11 (6)
DYFESM	flts	0 (0)	0 (0)	0 (0)	1 (0)	37 (22)	38 (22)	20 (12)	95 (56)
	ints	96 (40)	1 (0)	2 (1)	99 (41)	5 (3)	0 (0)	0 (0)	5 (3)
FLO52Q	flts	4 (2)	3 (1)	1 (0)	8 (3)	41 (24)	34 (20)	18 (10)	93 (53)
	ints	92 (40)	0 (0)	0 (0)	92 (40)	6 (4)	0 (0)	0 (0)	7 (4)
MDG	flts	13 (6)	7 (3)	0 (0)	20 (9)	38 (21)	23 (12)	19 (10)	80 (44)
	ints	78 (35)	0 (0)	2 (1)	80 (36)	20 (11)	0 (0)	1 (0)	20 (11)
QCD2	flts	6 (3)	0 (0)	1 (1)	8 (4)	43 (22)	17 (8)	14 (7)	74 (37)
	ints	86 (43)	2 (1)	5 (2)	92 (46)	18 (9)	4 (2)	4 (2)	26 (13)
TRACK	flts	14 (7)	10 (6)	6 (3)	30 (16)	37 (17)	23 (11)	19 (9)	79 (36)
	ints	68 (37)	1 (0)	1 (1)	70 (38)	17 (8)	1 (0)	2 (1)	21 (9)
TRFD	flts	0 (0)	0 (0)	0 (0)	0 (0)	37(23)	38 (23)	21 (13)	96 (59)
	ints	98 (39)	0 (0)	2 (1)	100 (39)	3 (2)	0 (0)	1 (0)	4 (2)

Table A.2: Operation breakdown as percentages

Prog.	Op Type	AU				DU			
		ops.	loads	stores	total	ops.	loads	stores	total
ADM	flts	1.130	0.775	0.323	2.228	19.465	12.613	11.222	43.299
	ints.	33.615	0.189	1.041	34.845	5.271	0.046	0.129	5.446
	tot.	34.744	0.965	1.363	37.072	24.736	12.658	11.351	48.745
DYFESM	flts	0.019	0.080	0.085	0.181	10.072	10.268	5.578	25.918
	ints.	18.481	0.232	0.381	19.094	1.433	0	0	1.437
	tot.	18.501	0.308	0.466	19.275	1.151	10.269	5.582	27.356
FLO52Q	flts	1.000	0.954	0.208	2.161	15.704	12.949	6.732	35.385
	ints.	26.202	0	0.065	26.267	2.413	0	0.056	2.470
	tot.	27.202	0.953	0.273	28.429	18.117	12.949	6.788	37.855
MDG	flts	7.747	3.875	0.003	11.626	26.886	16.014	13.441	56.342
	ints.	44.851	0.073	0.950	45.875	13.817	0.128	0.561	14.506
	tot.	52.599	3.949	0.953	57.501	40.704	16.142	14.002	70.849
QCD2	flts	3.467	0.097	0.630	4.194	24.064	9.298	7.959	41.322
	ints.	47.922	0.921	2.538	51.382	9.972	2.391	2.501	14.865
	tot.	51.389	1.018	3.168	55.577	34.036	11.690	10.461	56.188
TRACK	flts	1.394	1.078	0.601	3.074	3.173	1.988	1.674	6.836
	ints.	7.000	0.055	0.141	7.197	1.507	0.077	0.207	1.792
	tot.	8.394	1.133	0.743	10.271	4.680	2.066	1.882	8.628
TRFD	flts	0.018	0	0	0.018	30.504	31.153	17.420	79.078
	ints.	52.015	0.205	0.828	53.049	2.231	0.199	0.518	2.950
	tot.	52.034	0.205	0.828	53.068	32.736	31.352	17.939	82.029

Table A.3: Absolute count of operations ($\times 10^6$)

Appendix B

Derivation of β for Two Cache Models

This appendix shows how β can be derived for a write back and write through cache. The derivation assumes a fully interleaved pipelined main memory. After an initial access time of c_M cycles a word can be returned every cycle. The derivations are presented here to illustrate how β can be used for different latency hiding techniques. We use the equation for β given by

$$\begin{aligned} \beta &= \frac{\delta l}{c_M} \\ \textit{where} & \\ \delta l &= \delta c - (\overline{p} - c_m) \\ \textit{and} & \\ \overline{p} &= (1-w)[\alpha_r \mathsf{H_r} + (1-\alpha_r)\mathsf{M_r}] + w[\alpha_w \mathsf{H_w} + (1-\alpha_w)\mathsf{M_w}] \end{aligned}$$

B.1 β for a write back cache

For a write allocate policy the cost functions $\mathsf{H_r}$, $\mathsf{M_r}$, $\mathsf{H_w}$ and $\mathsf{M_w}$ are given by

$$\begin{aligned} \mathsf{H_r} &= c_m \\ \mathsf{H_w} &= c_m \\ \mathsf{M_r} &= (1+w_b)c_M + (1+w_b)(b-1) \\ \mathsf{M_w} &= (1+w_b)c_M + (1+w_b)(b-1) \end{aligned}$$

substituting into the above equation for β we get

$$\beta\delta c = \delta c - [(1-w)[\alpha_r c_m + (1-\alpha_r)[c_M + w_b c_M + (1+w_b)(b-1)]]$$

$$+w[\alpha_w c_m + (1-\alpha_w)[c_M + w_b c_M + (1+w_b)(b-1)]]] + c_m$$

$$= \delta c - [(1-w)[-(1-\alpha_r)c_m + (1-\alpha_r)c_M + (1-\alpha_r)[w_b c_M + (1+w_b)(b-1)]]$$
$$+w[-(1-\alpha_w)c_m + (1-\alpha_w)c_M + (1-\alpha_w)[w_b c_M + (1+w_b)(b-1)]]]$$

$$= \delta c - [(1-w)(1-\alpha_r)\delta c + w(1-\alpha_w)\delta c]$$
$$-[(1-w)(1-\alpha_r) + w(1-\alpha_w)][w_b c_M + (1+w_b)(b-1)]$$

$$= \beta^{ideal}\delta c - (1-\beta^{ideal})(w_b c_M + (1+w_b)(b-1))$$

$$\beta = \beta^{ideal} - \frac{(1-\beta^{ideal})}{\delta c}(w_b c_M + (1+w_b)(b-1))$$

B.2 β for a write through cache

For a no write allocate policy the cost functions $\mathsf{H_r}$, $\mathsf{M_r}$, $\mathsf{H_w}$ and $\mathsf{M_w}$ are given by

$$\begin{aligned} \mathsf{H_r} &= c_m \\ \mathsf{H_w} &= c_m + \delta c \\ \mathsf{M_r} &= c_M + (b-1) \\ \mathsf{M_w} &= c_M \end{aligned}$$

substituting into the above equation for β we get

$$\beta\delta c = \delta c - [(1-w)[\alpha_r c_m + (1-\alpha_r)(c_M + (b-1))]$$
$$+w[\alpha_w(c_m + \delta c) + (1-\alpha_w)c_M]] + c_m$$

$$= \delta c - [(1-w)[-(1-\alpha_r)c_m + (1-\alpha_r)c_M + (1-\alpha_r)(b-1)]$$
$$+w[-(1-\alpha_w)c_m + \alpha_w\delta c + (1-\alpha_w)c_M]]$$

$$= \delta c - (1-w)(1-\alpha_r)\delta c - (1-w)(1-\alpha_r)(b-1)$$
$$-w(1-\alpha_w)\delta c - w\alpha_w\delta c$$

$$\beta = \beta^{ideal} - w\alpha_w - \frac{(1-w)(1-\alpha_r)(b-1)}{\delta c}$$

Appendix C

Simulation Sampling

Section 5.3.4 showed that a full execution of the program is sometimes necessary to capture all of a program's behaviour. However, the experimental run time for a full simulation of a program could be as long as 12 hours, making it prohibitively expensive for detailed studies of decoupled machines. To solve this problem a *sampling technique* was developed that reduced experimental execution time while capturing different aspects of program behaviour. The technique worked by identifying *sampling sections* in the program that needed to be simulated and then switching the simulator on and off between these sections.

Each program was run to completion and the number of loads executed per 100,000 program statements were counted. The load count profiles for each of the programs is shown in Figures C.1 to C.7. It can be observed that the profiles show in most cases different regions of periodic behaviour For example, in Figure C.5 the graph for QCD2 shows three distinct regions. The sampling technique works by switching the simulator on for each distinctive region but only for the length of the periodic behaviour; this length is known as the sampling section. Table C.1 shows the sampling sections for each of the programs. The column labelled *sampled sections* shows the range of program statements for which the simulator was switched on. For example, in ADM the simulator was switched on and off when, respectively, the 42nd and 56th million program statement was reached.

Other modelling techniques reduce experiment execution times by a *continuous* simulation of the first N operations (where N is some large number) or ensure that the distribution of simulated operations is the same as the whole program. However neither of these solve the problem of modelling the varying behaviour in a program. To evaluate the benefits of the sampled simulation, a comparison was made with a full and continuous simulation. Since one of the main interests is latency hiding, a sensible comparative measurement is the average perceived load latency. We also want to make sure that the the ratio of AU to DU operations

is comparable to the full simulation. The simulations were run with 60 cycle memory differential.

Table C.2 shows the program characteristics for the different types of simulation; *F, S, C* denotes a full, sampled and continuous simulation, respectively; the number in brackets denotes the factor by which the number of statements executed in the continuous simulation is greater than the sampled. It can be seen that in programs with complex behaviour, QCD2 and TRACK, the sampling technique provides considerable benefits over a continuous simulation approach. For these programs sampling is shown to be within 6% of the $\overline{pll}$ of the full simulation at a lower execution cost. In QCD2 a 300% increase in execution time for a continuous simulation fails to yield an $\overline{pll}$ value better than the sampled run.

All the experiments in this section were run on the dual stream architecture (see Figure 3.2) with unlimited resources and a floating point latency of 5 cycles.

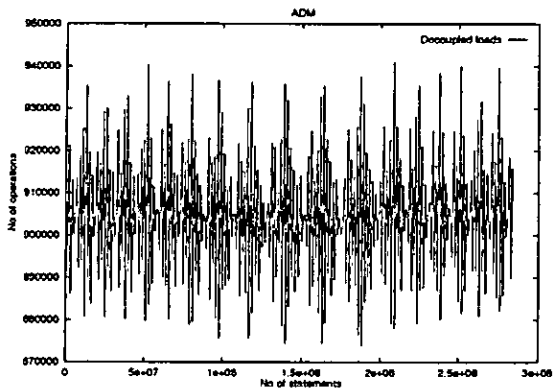


Figure C.1: ADM profile of load operations

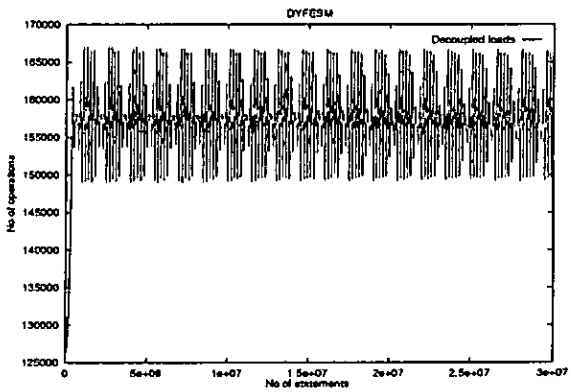


Figure C.2: DYFESM profile of load operations

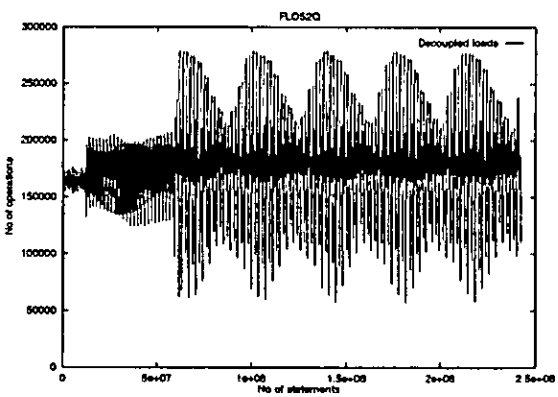


Figure C.3: FLO52Q profile of load operations

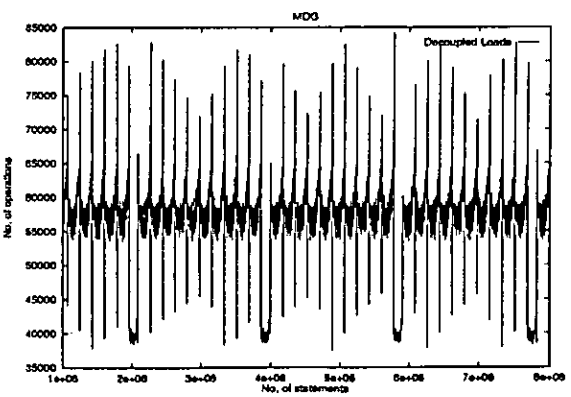


Figure C.4: MDG profile of load operations

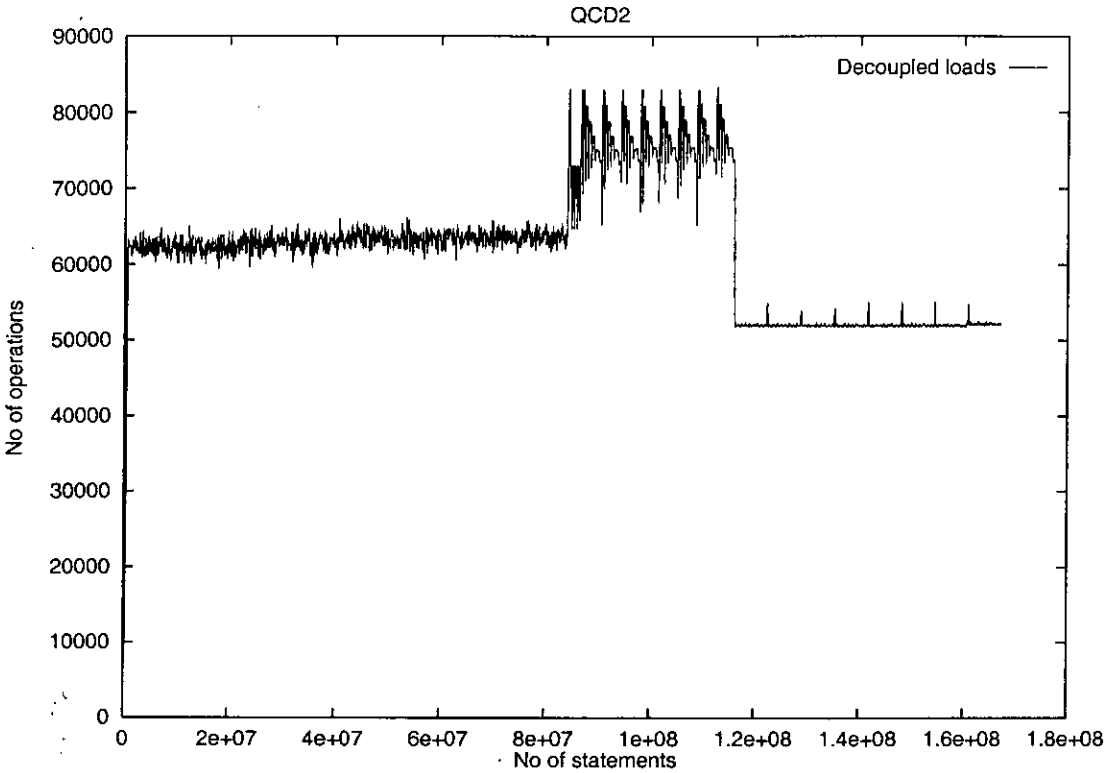


Figure C.5: QCD2 profile of load operations

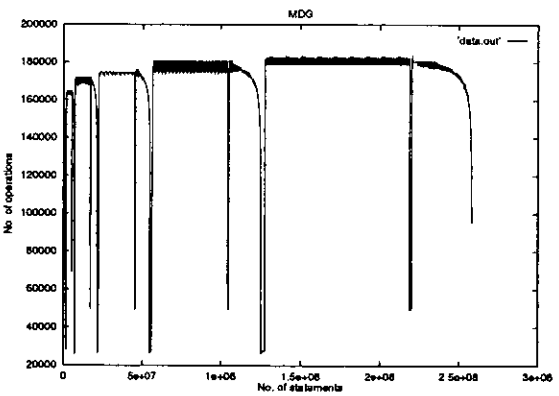


Figure C.6: TRFD profile of load operations

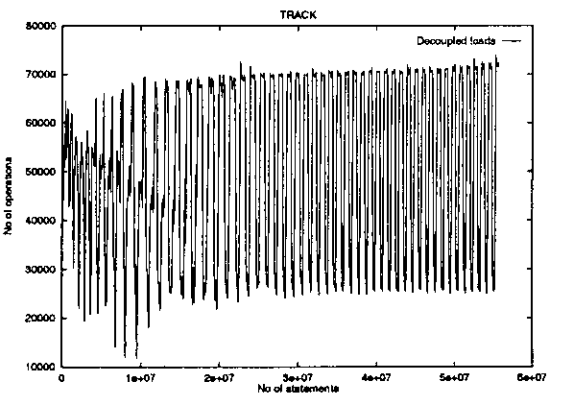


Figure C.7: TRACK profile of load operations

Program	No. of Section	Sampled sections	
		From ($\times 10^6$)	To ($\times 10^6$)
ADM	1	42	56
DYFESM	1	20	26.64
FLO52Q	1	6	7.4
	2	35	37
	3	65	69
MDG	1	19.5	22.7
QCD2	1	10	12.3
	2	30	32.3
	3	60	64.6
	4	90	94
	5	122	128
TRACK	1	3	4.7
	2	9	11
	3	36	37
TRFD	1	40	60

Table C.1: Program sampled sections

Program	Length of sim.	*pll* (cycles)	Operations ($\times 10^6$)			Time
			AU (%)	DU (%)	Loads (%)	(hrs:mins)
ADM	F	42	754.1 (43)	990.3 (57)	276.7 (16)	13:40
	S	43	37.1 (43)	48.7 (57)	13.6 (16)	0:50
	C (x 1)	41	37.0 (43)	48.7 (57)	13.6 (16)	0:40
DYFESM	F	51	544.8 (41)	772.6 (59)	298.8 (23)	10:30
	S	51	19.3 (41)	27.4 (59)	10.6 (23)	0:27
	C (x 1)	50	19.3 (41)	27.2 (59)	10.5 (23)	0:25
FLO52Q	F	36	941.7 (43)	1254.5 (57)	462.6 (21)	19:40
	S	36	28.4 (43)	37.9 (57)	13.9 (21)	0:36
	C (x 1)	36	27.8 (44)	35.1 (56)	12.9 (20)	0:36
MDG	F	39	2963 (38)	4917 (62)	1294 (16)	13:40
	S	40	45.5 (38)	77.1 (62)	20.1 (16)	0:25
	C (x 1)	41	49.5 (38)	82.1 (62)	21.6 (16)	0:21
QCD2	F	49	484.0 (50)	489.6 (50)	110.3 (11)	2:00
	S	49	55.6 (50)	56.2 (50)	12.7 (11)	0:27
	C (x 1)	44	47.3 (47)	52.6 (53)	12.5 (13)	0:13
	C (x 2)	44	94.7 (47)	105.9 (53)	25.3 (13)	0:26
	C (x 4)	44	189.5 (47)	213.3 (53)	51.2 (13)	0:53
	C (x 8)	48	356.5 (48)	386.6 (52)	93.1 (13)	1:30
TRACK	F	51	117.2 (48)	125.5 (52)	37.7 (16)	2:20
	S	54	10.3 (54)	8.6 (46)	3.2 (17)	0:17
	C (x 1)	59	12.3 (61)	7.9 (39)	4.1 (20)	0:13
	C (x 2)	58	23.1 (58)	16.4 (42)	7.5 (19)	0:25
	C (x 4)	56	42.6 (55)	35.2 (45)	13.6 (17)	0:55
	C (x 8)	52	80.5 (50)	81.3 (50)	25.8 (16)	1:23

Table C.2: Program characteristics for different length of simulation (F=full, S=sampled, C=continuous)

Appendix D

Balanced Configurations : Additional Data

This appendix presents the data for the benchmark programs not shown in Chapter 6. The graphs show the variation in *normalised speedup* for different configurations of AU and DU issue width.

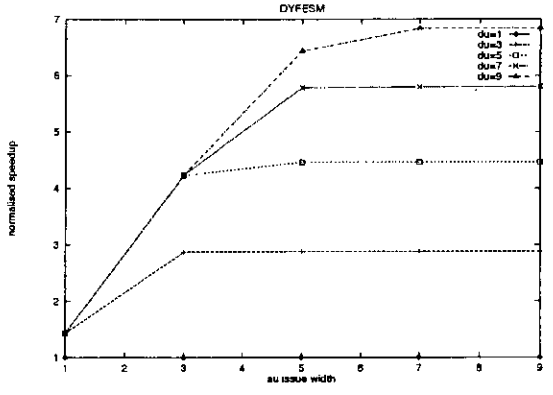


Figure D.1: DYFESM MD=0

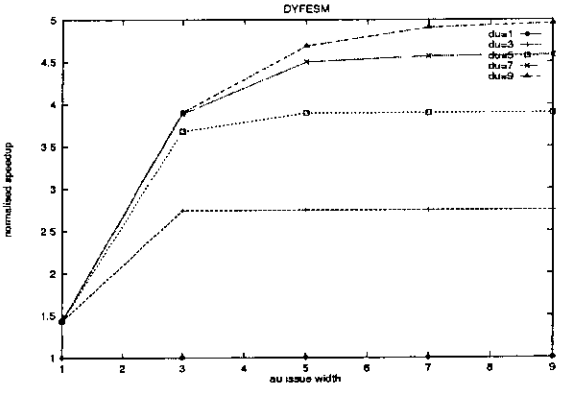


Figure D.2: DYFESM MD=60

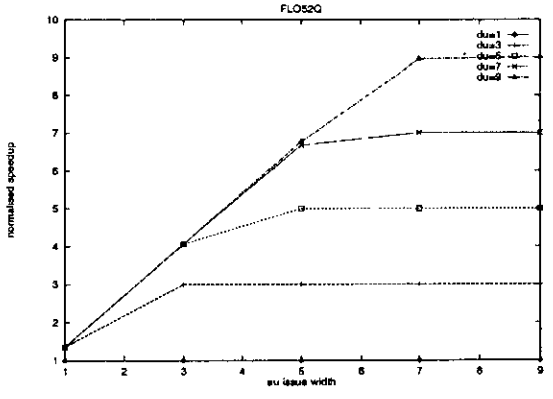


Figure D.3: FLO52Q MD=0

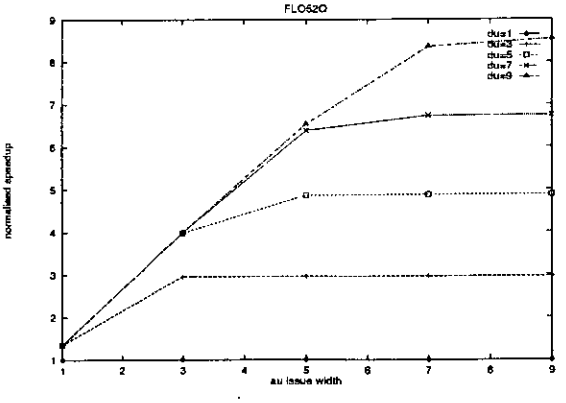


Figure D.4: FLO52Q MD=60

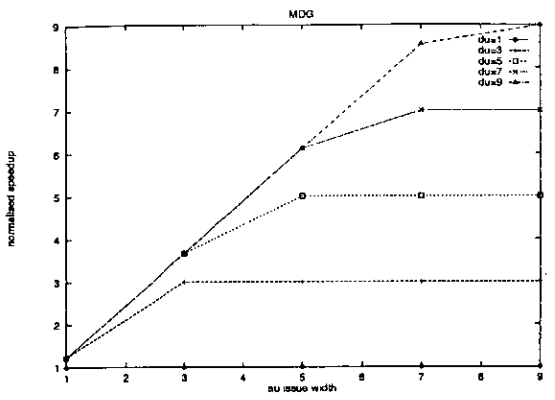


Figure D.5: MDG MD=0

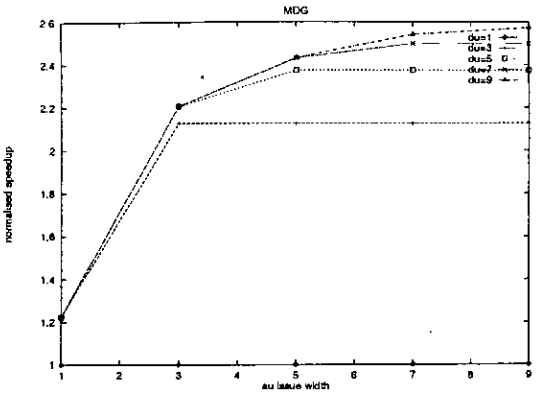


Figure D.6: MDG MD=60

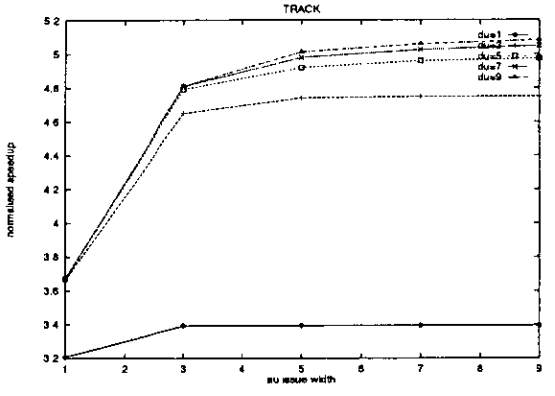


Figure D.7: TRACK MD=0

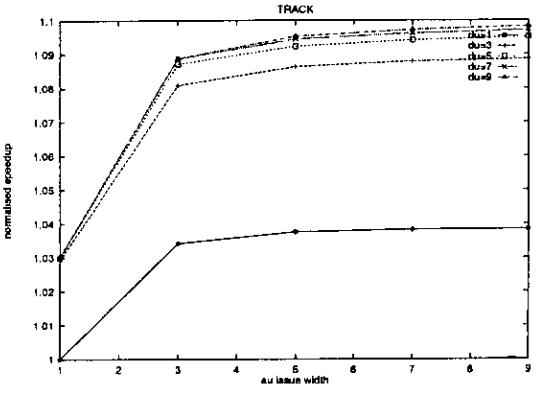


Figure D.8: TRACK MD=60

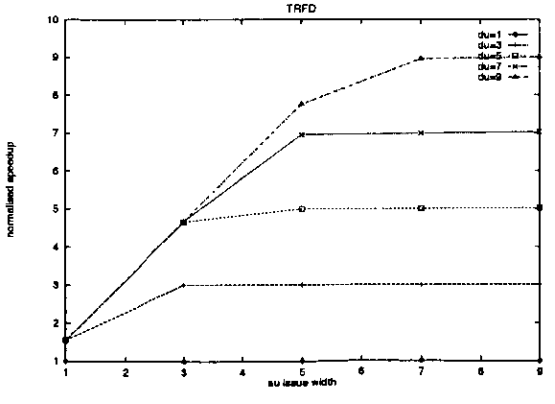


Figure D.9: TRFD MD=0

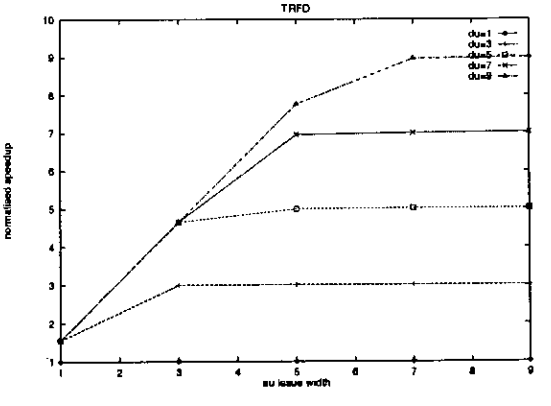


Figure D.10: TRFD MD=60

Appendix E

Reducing Hardware Complexity : Additional Data

This appendix presents the data for the benchmark programs not shown in Chapter 8. The graphs show the variation equivalent window ratio with window size and memory differential (MD).

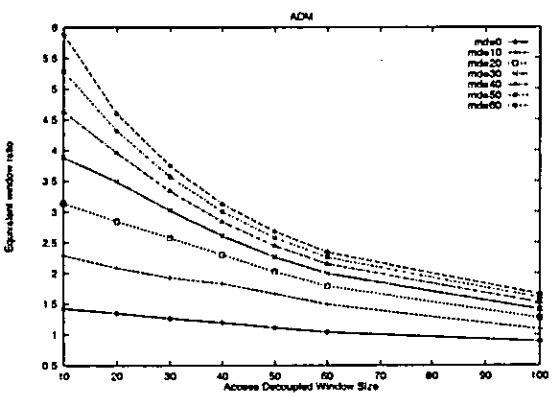


Figure E.1: ADM

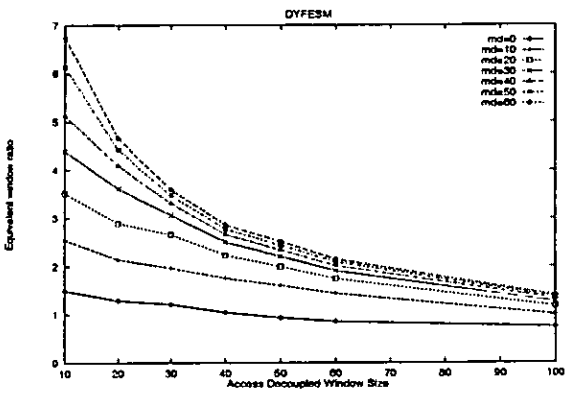


Figure E.2: DYFESM

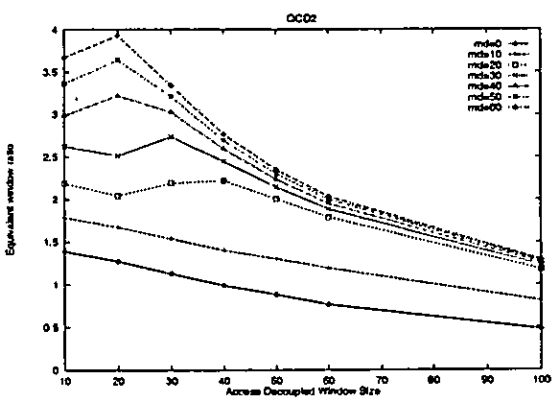


Figure E.3: QCD2

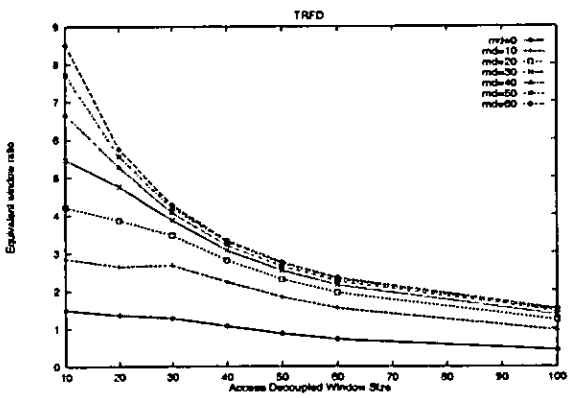


Figure E.4: TRFD

Appendix F

Sensitivity Analysis : Additional Data

This appendix shows additional data in support Chapter 9. The graphs for the equivalent window ratio for ADM, DYFESM, QCD2 and TRFD are shown in Figures F.1, F.2, F.3 and F.4, respectively.

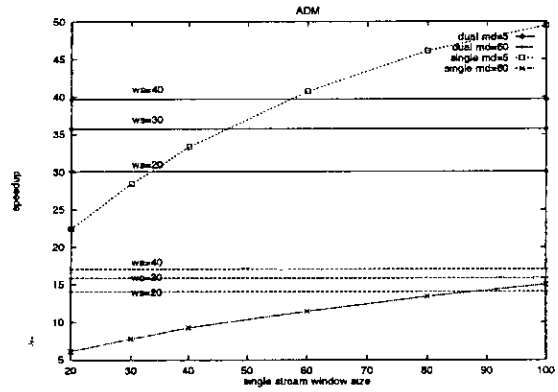


Figure F.1: ADM speedup

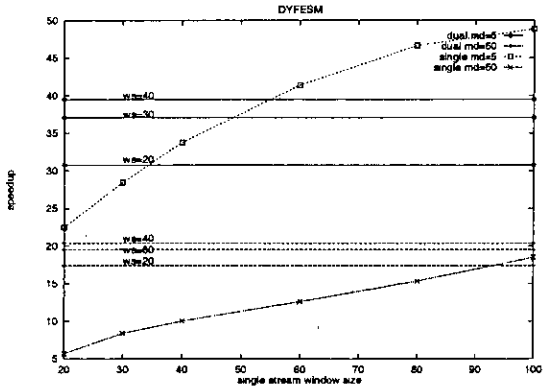


Figure F.2: DYFESM speedup

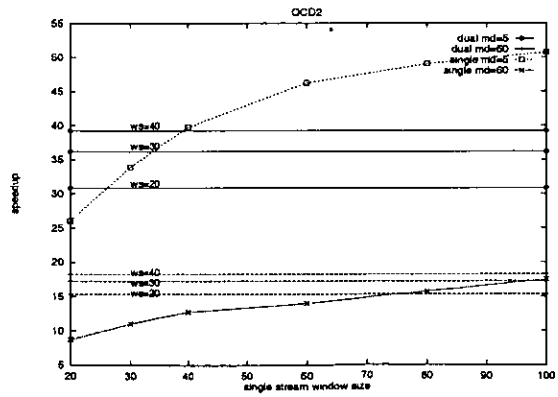


Figure F.3: QCD2 speedup

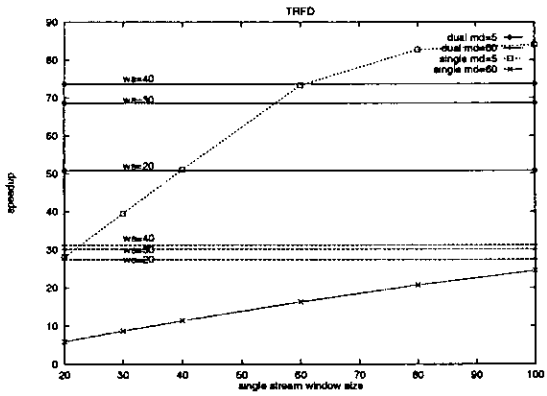


Figure F.4: TRFD speedup

Bibliography

[1] A.V. Aho and R. Sethi J.D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley publishing company, 1986.

[2] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Coversion of Control Dependence to Data Dependence. In *POPL*, pages 177–189, Jan. 1983.

[3] D. Anderson, F. Sparacio, and R. Tomasulo. The IBM System 360 Model 91: Machine Philosophy and Instruction Handling. *IBM Journal of Research and Development*, pages 8–24, Jan. 1967.

[4] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. Technical Report UCB/CSD-93-781, University of California, Berkeley, California 94720, 1993.

[5] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1st edition, 1988.

[6] G. Beck, D.W.L., and T.L. Anderson. The Cydra 5 Minisupercomputer: Architecture an Implementation. *The Journal of Supercomputing*, 7(1/2):143–180, 1993.

[7] A. Berrached, L.D. Coraor, and P.T. Hulina. A Decoupled Access/Execute Architecture for Efficient Access of Structured Data. In *Proc. of the 26th Hawai Int. Conf. on System Sciences*, volume 1, pages 438–47, Los Alamitos, CA, USA, Jan 1993. IEEE Press.

[8] D. Bhandarkar and J. Ding. Performance Characterisation of the Pentium Pro Processor. In *Proc. of the 3rd Int. Symp. on High Performance Computer Architecture*, San Antonio, Texas, USA., Feb. 1997. IEEE.

[9] P. Bird, A. Rawsthorne, and N.P. Topham. The Effectiveness of Decoupling. In *Proc. Int. Conf. on Supercomputing*, Tokyo, Japan, May 1993.

[10] W. Blume and R. Eigenmann. Performance Analysis of Parallelising Compilers on the Perfect Benchmark Programs. *IEEE Trans. on Parallel and Distributed Systems.*, 3(6):643–656, Nov. 1992.

[11] W. Blume and R. Eigenmann. Symbolic Analysis Techniques for the Effective Parallelisation of the Perfect Benchmarks. CSRD 1332, Center for Supercomputing Research and Development., University of Illinois at Urbana-Champaign, Urbana, Illinois 61801, Jan. 1994.

[12] S.E. Breach, T.N. Vijaykumar, and G.S. Sohi. The Anatomy of the Register File in a Multiscalar Processor. In *27th Int. Symp. on Microarchitecture (MICRO-27)*, 1994.

[13] D. Burger, J. Goodman, and A. Kagi. The Declining Effectiveness of Dynamic Caching for General Purpose Microprocessors. Technical Report 1261, Uni. Wisconsin-Madison, Jan. 1995.

[14] M. Burke and R. Cytron. Interprocedural Dependence Analysis and Parallelisation. In *Proc. of SIGPLAN 1986 Symp. on Compiler Construction.*, pages 162–175, 1986.

[15] M. Byler, J. Davis, C. Huson, B. Leasure, and D. Padua. Multiple version loops. In *Proc. of the 1987 Intenational Conf. on Parallel Processing*, pages 312–318, Aug 1987.

[16] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *4th ASPLOS*, pages 40–52, Santa Clara, California, Apr. 1991.

[17] T. Chen and J. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proc. of the 21st Ann. Int. Symposioum on Computer Architecture.*, pages 223–232, Los Alamitos, CA, USA, April 1994. IEEE, ACM, IEEE Computer Press.

[18] Tzi-cker Chiueh. Sunder : A Programmable Hardware Prefetch Architecture for Numerical Loops. In *Proc. Supercomputing '94*, pages 488–497, Los Alamitos, CA, USA, Nov 1994. IEEE Comput. Soc., ACM , SIAM, IEEE Comput. Soc. Press.

[19] Z. Cventanovic and D. Bhandarkar. Performance Characterisation of the Alpha 21164 Microporocessor using TP and SPEC Workloads. In *2nd Int. Symp. HPCA*, pages 270–280, Feb. 1996.

[20] F. Dahlgren and M. Dubois. Sequential Hardware Prefetching in Shared Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7), Jul. 1995.

[21] J. Davidson and S. Jinturkar. Improving Instruction Level Parallelism by Loop Unrolling and Dynmaic Memory Disambiguation. In *28th Int. Symp. on Microarchitecture (MICRO-28)*, pages 125–32, Nov. 1995.

[22] J. Dehnert and R. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 7:181–227, 1993.

[23] R. Eigenmann, J. Hoeflinger, L. Zhiyuan, and D. Padua. Experience in the Automatic Parallelisation of Four Perfect-Benchmark Programs. CSRD 1193, Center for Computing Research and Development., University of Illinois at Urbana-Champaign, Urbana Illinois., 1992.

[24] J.E. Smith et al. The ZS-1 Central Processor. In *2nd ASPLOS*, October 1987.

[25] M. Berry et al. The Perfect Club Benchmarks, Effective Performance Evaluation of Supercomputers. Techreport 827, CSRD, University of Illinois, Urbana-Chmpaign, Urbana, Illinois., May 1989.

[26] U. Banerjee et al. Automatic Program Parallelisation. *Proc. of the IEEE*, 81(2):211–243, Feb. 1993.

[27] A. Scott *et al.*. Four-Way Superscalar PA-RISC Processors. *Hewlett Packard Journal*, pages 1–9, Aug. 1997.

[28] C. Kozyakis *et al.*. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, 30(9):75–78, 1997.

[29] K. Olukotun *et al.*. The Case for a Single Chip Multiprocessor. In *ASPLOS VII*, Oct. 1996.

[30] K.I. Farkas *et al.*. Memory System Design Considerations for Dynamically Scheduled Processors. In *24th Int. Ann. Symp. on Computer Architecture (ISCA-24)*, June 1997.

[31] R. Alverson *et al.*. The Tera Computer System. *Proc. Int. Conf. on System Science*, pages 1–6, June 1990.

[32] S. Adve *et al.*. Changing Interaction of Compiler and Architecture. *IEEE Computer*, 30(12):51–57, Dec. 1997.

[33] S. Mckee *et al.*. Design and Evaluation of Dynamic Access Ordering Hardware. In *10th Int. Conf. on Supercomputing (ICS'96)*, May 1996.

[34] W. Blume *et al.* Automatic Detection of Parallelism: A Grand Challenge for High Performance Computing. CSRD 1348, Center for SuperComputing Research and Development., University of Illinois at Urbana-Champaign, 1308 W.Main St, Urbana, Illinois, July 1994.

[35] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *MICRO-30*, Dec. 1997.

[36] K.I. Farkas and N.P. Jouppi. Comlpexity/Peformance Tradeoffs with Non-Blocking Loads. In *Int. Symp. on Computer Architecture (ISCA-21)*, pages 211–222, 1997.

[37] M.K. Farrens and A.R. Pleszkun. Implementation of the PIPE Processor. *IEEE Computer*, pages 65–70, Jan 1991.

[38] M.K. Farrens, P.Ng, and P.Nico. A Comparison of Superscalar and Decoupled Access/Execute Architectures. In *Int. Symp. on Microarchitecture (MICRO-26)*, Dec. 1993.

[39] M. Franklin and G. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism. In *19th Int. Symp. on Computer Architecture*, 1992.

[40] M. Franklin and G.S. Sohi. ARB: A Hardware Mechanism for Dynamic Rordering of Memory Reference. *IEEE Trans. on Computers*, 45(5), May 1996.

[41] J.W.C. FU and J.H. Patel. Data Prefetching Strategies for Vector Cache Memories. In *Proc. 5th Int. Parallel Processing Symp.*, pages 555–560, Los Alamitos, CA, USA, April-May 1991. IEEE Computer Society Press.

[42] L. Gwennap. Digital 21264 sets new standard. *Microprocesor Report*, 10(14), Oct. 1996.

[43] M. Hall, J. Mellor-Crummey, A. Carle, and R. Rodriguez. FIAT: A Framework for Interprocedural Analysis and Transformation. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, 6th Int. Workshop*, pages 522–545. Springer-Verlag, Aug 1993.

[44] L. Hammond, B. Nayfeh, and K. Olukotun. A Single Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, Sept. 1997.

[45] J. Hennessy and D. Patterson. *Computer Architecture A Qualitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[46] A. Huang and J. Shen. A Limit Study of Local Memory Requirements Using Value Reuse Profiles. In *28th Int. Symp. on Microarchitecture (MICRO-28)*, Nov. 1995.

[47] G.P. Jones and N.P. Topham. A comparison of data prefetching on an access decoupled and superscalar machine. In *Int. Sym. on Microarchitecture (MICRO-30)*, pages 65–71. IEEE Computer Society, Dec. 1997.

[48] G.P. Jones and N.P. Topham. The Effect of Restricted Instruction Issue Width on an Access Decoupled Architecture. In *Parco97*, pages 665–673, Sept. 1997.

[49] G.P. Jones and N.P. Topham. A Limitation Study into Access Decoupling. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, pages 1102–1111. University of Passau, Germany, Springer, Aug. 1997.

[50] N.P. Jouppi. Improving Direct Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers. In *17th Int. Symp. on Computer Architecture (ISCA-17)*, pages 364–373, May 1990.

[51] G. Kemp and M. Franklin. PEWs: A Decentralised Dynamic Scheduler for ILP Processing. In *Proc. Int. Conf. on Parallel Processing*, volume 1, pages 239–246, 1996.

[52] A. Klaiber and H. Levy. Architecure for Software Controlled Data Prefetching. In *ISCA-18*, pages 43–63, May 1991.

[53] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organisation. In *8th Int. Symp. on Computer Architecture*, pages 81–87, 1981.

[54] M. Kumar. Measuring Parallelism in Computation Intensive Scientific/Engineering Applications. *IEEE Trans. on Computers*, 37(9):1088–1098, Sept. 1988.

[55] L. Kurian, P.T. Hulina, and L.D. Coraor. Memory Latency Effects in Decoupled Architectures. *IEEE Trans. on Computers*, 43(10):1129–39, Oct. 1994.

[56] L. Kurian, V. Reddy, P.T. Hulina, and L.D. Coraor. A Comparative Evaluation of Software Techniques to Hide Memory Latency. In T. Mudge and B.D. Shriver, editors, *Proc. of the 28th Hawaii Internatonal Conference on System Sciences*, volume 1, pages 229–38, Los Alamitos, CA, USA, Jan 1995. IEEE Computer Society Press.

[57] M. Lam and R.P. Wilson. Limits of Control Flow on Parallelism. In *Proc. 19th Int. Symp. on Computer Architecture.*, pages 46–57, May 1992.

[58] W. Mangione-Smith, S.G. Abraham, and E.S. Davidson. A Performance Comparison of the IBM RS6000 and the Astronautics ZS-1. *IEEE Computer*, pages 39–46, Jan. 1991.

[59] S. McFarling. Combing Branch Predictors. Technical Report TN-36, Digital WRL, June 1993.

[60] S. Mckee and W. Wulf. A Memory Controller for Imporved Performance of Streamed Computations on Symmetric Multiprocessors. In *10th Int. Parallel Processing Symp. (IPPS'96)*, Apr. 1996.

[61] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic Speculation and Synchronisation of Data Dependencies. In *ISCA-24*, Jun. 1997.

[62] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *5th ASPLOS*, pages 62–73, Oct. 1992.

[63] D. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Comminications of the ACM*, 29(12):1184–1201, Dec. 1986.

[64] S. Palacharla, N.P. Jouppi, and J.E. Smith. Complexity-Effective Superscalar Processors. In *24th Ann. Int. Symp. on Computer Architecture*, 1997.

[65] P. Petersen. *Evaluation of Programs and Parallelising Compilers using Dynamic Analysis Techniques.* PhD thesis, University of Illinois, Urbana-Champaign, Illinois, 1993.

[66] P.M. Petersen. and D.A. Padua. Evaluation of Parallelsing Compilers. CSRD 1279, Center for Supercomputing Research and Development., University of Illinois at Champaign-Urbana, Urbana, Illinois, 61801, 1992.

[67] W. Pugh. A Practical Algorithm for Exact Dependence Analysis. *Communications of the ACM*, 35(8):102–114, Aug. 1992.

[68] J. Rivers, G. Tyson, and E. Davidson. On High Bandwidth Data Cache Dseign for Multi-Issue Processors. In *MICRO-30*. IEEE, Dec 1997.

[69] Z. Shen, Z. Li, and P-C. Yew. An Empirical Study on Array Subscripts and Data Dependences. In *1989 Int. Conf. on Parallel Processing*, pages II–145–II–152, 1989.

[70] J.E. Smith. Decoupled Access/Execute Computer Architectures. *ACM Trans. on Computer Systems*, 2(4):289–308, Nov. 1984.

[71] J.E. Smith. A Simulation Study of Decoupled Architecture Computers. *IEEE Trans. on Computers*, C-35(8):692–701, Aug. 1986.

[72] J.E. Smith. Dynamic scheduling and the astronautics zs-1. *IEEE Computer*, pages 21–35, July 1989.

[73] J.E. Smith and G.S.Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of IEEE*, 83(12):1609–1624, Dec. 1995.

[74] G. Sohi and M. Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *4th ASPLOS*, Apr. 1991.

[75] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *22nd Int. Symp. on Computer Architecture*, 1995.

[76] S. Song, M. Denman, and J. Chang. The PowerPC 604 RISC Microprocessor. *IEEE Micro*, 14(5):8–17, Oct. 1994.

[77] S.Palacharla and R.E. Kessler. Evaluating Sream Buffers as a Secondary Cache Replacement. In *Proc. of the 21st Ann. Int. Symp. on Computer Architecture.*, pages 24–33, Los Alamitos, CA, USA, Apr. 1994. IEEE, ACM, IEEE Computer Society Press.

[78] K. Sundararaman and M. Franklin. Multiscalar Execution along a Single Flow of Control. In *Int. Conf. on Parallel Processing*, volume 1, 1997.

[79] N.P. Topham, A. Rawsthorne, C.E. McLean, M.J.R.G. Mewissen, and P.Bird. Compiling and Optimising for Decoupled Architectures. In *Proc. of Supercomputing '95*, San Diego, Dec. 1995. ACM press.

[80] G. Tyson, M. Farrens, and A.R. Pleszkun. MISC : A Mutiple Instruction Stream Computer. In *Int. Sym. on Microarchitecture (MICRO-25*, Portland, Oregon, Dec 1-4 1992.

[81] G.S. Tyson. *Evaluation of a Scalable Decoupled Microprocessor Design.* PhD thesis, University of California Davis, 1997.

[82] S. Vajapeyam, G.S. Sohi, and W-C Hsu. An Empirical Study of the CRAY YMP Processor using the PERFECT Club Benchmarks. In *Proc. of the 1991 ACM Int. Conf. on Supercomputing*, pages 170–179, New York, 1991. ACM, ACM press.

[83] D. Wall. Limits of Instruction Level Parallelism. In *4th Int. Conf. on ASPLOS*, pages 8–11, Apr 1991.

[84] D. Wall. Speculative Execution and Instruction Level Parallelism. Technical Report TN-42, Digital WRL, Mar. 1994.

[85] T. Williams, N. Patkar, and G. Shen. SPARC64: A 64-b 64 Active Instruction Out-of-Order Execution MCM Processor. *IEEE Journal of Solid-State Circuits*, 30(11):1215–26, Nov. 1995.

[86] K. Wilson, K. Olukotun, and M. Rosenblum. Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors. In *23rd Int. Sym. on Computer Architecture (ISCA-23)*, pages 147–157, May 1996.

[87] M. Wolfe. *Optimising Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. Pitman, 1 edition, 1989.

[88] W.M. Wulf. An Evaluation of the WM Architecture. In *Proceedings of the Intenational Symposium on Computer Architecture*, Palo Alto, CA, Oct. 1987.

[89] Wm. Wulf and S. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, Mar. 1995.

[90] Wm A. Wulf. An Evaluation of the WM Architecture. In *Proc. Int. Symp. on Computer Architecture*, pages 382–390, Gold Coast, Australia, May 1992.

[91] K.C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE micro*, 16(2):28–41, April 1996.

[92] H. Zima. *Supercompilers for Parallel and Vector Computers.* ACM press. Addison-Wesley, 1 edition, 1990.